

***CSE1303 Part A
Data Structures and Algorithms
Summer Semester 2003***

Lecture A17/18 – Revision

Kymerly Fergusson

Lecture Overview

- Subject overview
 - Topics covered this semester.
- Exam overview
 - Types of questions found in Part A of the exam.
- Exam hints and resources
 - How to deal with exams (resources, and hints for during the exam).
 - How to prepare.
- Revision
- Going further
 - Some examples you may find useful.

2

Subject Overview

- Basic C
 - Data types
 - 1D and multidimensional arrays
 - Strings
 - I/O & File I/O
 - Structures and **typedef**
 - Dynamic memory
 - Pointers

3

Subject Overview

- ADTs
 - Stacks
 - Array implementation
 - Linked implementation
 - Push
 - Pop
 - Initialise
 - Check empty/full

4

Subject Overview

- ADTs
 - Queues
 - Array implementation
 - Linked implementation
 - Append
 - Serve
 - Initialise
 - Check empty/full

5

Subject Overview

- ADTs
 - Singly linked list
 - Array implementation
 - Linked implementation
 - Insert
 - Delete
 - Search
 - Initialise
 - Check empty/full
 - Traversal

6

Subject Overview

- ADTs
 - Doubly linked list
 - Linked implementation
 - Insert (not C code)
 - Delete (not C code)
 - Search (not C code)
 - Initialise
 - Traversal (not C code)

7

Subject Overview

- ADTs
 - Trees
 - Parse Trees/Expression Trees
 - Prefix/Infix/PostFix
 - Binary Trees and Binary Search Trees
 - Insert
 - Delete
 - Search
 - Initialise
 - PreOrder, InOrder, PostOrder Traversal

8

Subject Overview

- ADTs
 - Hashtables
 - Hash function
 - Insert
 - Delete (chaining)
 - Search
 - Initialise
 - Collision resolution (chaining, linear probing)

9

Subject Overview

- Algorithms
 - Searching
 - Linear search (arrays, lists, hashtable)
 - Binary search (arrays)
 - Recursion
 - Direct/Indirect
 - Unary
 - Binary
 - Complexity

10

Subject Overview

- Algorithms
 - Sorting
 - Insertion sort (array)
 - Selection sort (array)
 - Binary Tree sort
 - Mergesort (array)
 - Quicksort (array)

11

Exam Overview

- 0.5 of the 3 hour exam is for Part A
- Typical types of questions:
 - Multiple choice
 - Short Answer
 - Write a structure or a small piece of C
 - Write a short answer to a question
 - Long answer
 - Code a solution to a problem
 - Write an algorithm for a problem
 - Fill in diagrams/missing code
- We will go over the sample exam next lecture.

12

Resources for Exams

- Monash Community services selfhelp information for exams:
 - [Exam Skills - Clue Words](#)
 - [Exam Taking Techniques](#)
 - [Exam Analysis](#)
 - [Effective Skills in Examinations](#)
 - [How to Survive Exam Weeks](#)
 - [Preparing for Tests and Exams](#)
 - [Final Exam Review Checklist](#)
 - [Examination Room Techniques](#)
 - [General Exam taking Hints](#)
 - [How to keep Calm during Tests](#)
 - [Test Anxiety Scale](#)
- All these resource pages found at:
 - <http://www.adm.monash.edu.au/commserv/counselling/selfhelp.htm>

13

Exam Preparation Hints

- Revise all of the lecture notes
- Do all of the tutorial questions
- Revise and finish all of the pracs (you gain better understanding doing the bonus questions)
- Do the suggested reading at the end of all lectures
- Do the suggested additional exercises in the tutorials
- Attempt the practice exam
- Revise your tests (look at them at the general office)
- Try to implement the algorithms that you haven't already done in the pracs
- Prepare questions for the last tutorial
- Come with questions to consultation with the lecturers (*additional hours are advertised closer to the exam date*)

14

During the Exam

- **Read the questions carefully!**
- If you don't know what a question means, ask the lecturer.
- Don't get stuck on one question – move onto something else.
- Plan your time – don't spend it all on a couple of questions.
- Do the easy questions first.
- Attempt all questions.
- Don't use whiteout! (And don't erase your workings)
- Check you have answered all the questions.

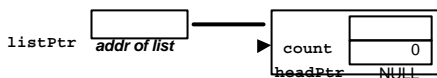
15

Revision – Linked Lists

- Operations:
 - Initialise
 - Set position (step to a position in the list)
 - Insert
 - Search
 - Delete
 - Make a new node

16

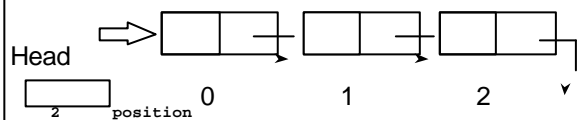
Initialise List



```
void initialiseList(List* listPtr)
{
    listPtr->headPtr = NULL;
    listPtr->count = 0;
}
```

17

Set Position

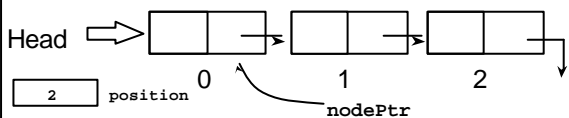


```
Node* setPosition(const List* listPtr, int position){
    int i;
    Node* nodePtr = listPtr->headPtr;

    if (position < 0 || position >= listPtr->count) {
        fprintf(stderr, "Invalid position\n");
        exit(1);
    }
    else {
        for (i = 0; i < position; i++) {
            nodePtr = nodePtr->nextPtr;
        }
    }
    return nodePtr;
}
```

18

Set Position

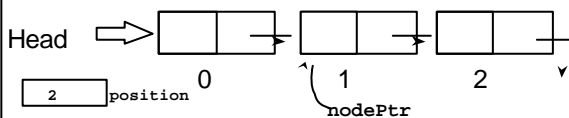


```
Node* setPosition(const List* listPtr, int position){
    int i;
    Node* nodePtr = listPtr->headPtr;

    if (position < 0 || position >= listPtr->count) {
        fprintf(stderr, "Invalid position\n");
        exit(1);
    }
    else {
        for (i = 0; i < position; i++) {
            nodePtr = nodePtr->nextPtr;
        }
    }
    return nodePtr;
}
```

19

Set Position

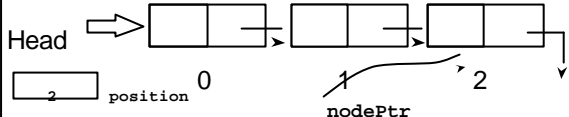


```
Node* setPosition(const List* listPtr, int position){
    int i;
    Node* nodePtr = listPtr->headPtr;

    if (position < 0 || position >= listPtr->count) {
        fprintf(stderr, "Invalid position\n");
        exit(1);
    }
    else {
        for (i = 0; i < position; i++) {
            nodePtr = nodePtr->nextPtr;
        }
    }
    return nodePtr;
}
```

20

Set Position

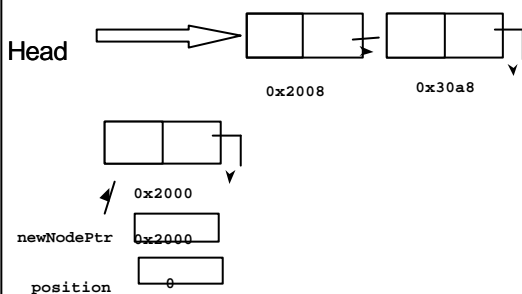


```
Node* setPosition(const List* listPtr, int position){
    int i;
    Node* nodePtr = listPtr->headPtr;

    if (position < 0 || position >= listPtr->count) {
        fprintf(stderr, "Invalid position\n");
        exit(1);
    }
    else {
        for (i = 0; i < position; i++) {
            nodePtr = nodePtr->nextPtr;
        }
    }
    return nodePtr;
}
```

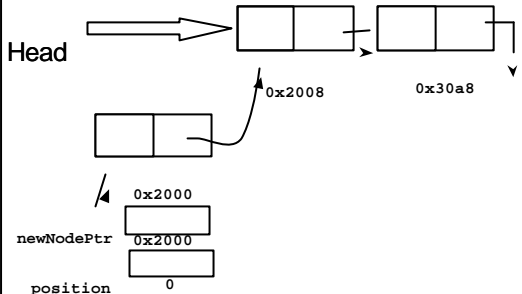
21

Inserting – Start of List



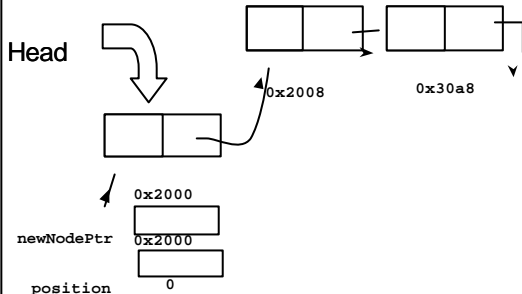
22

Inserting – Start of List



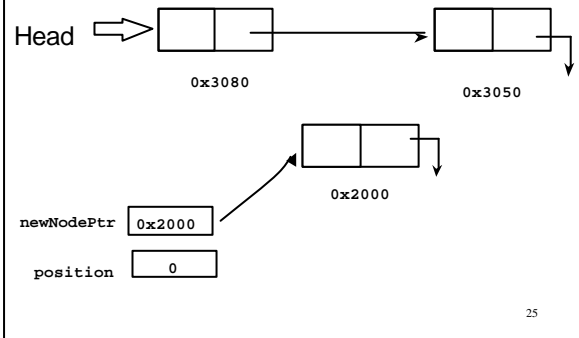
23

Inserting – Start of List

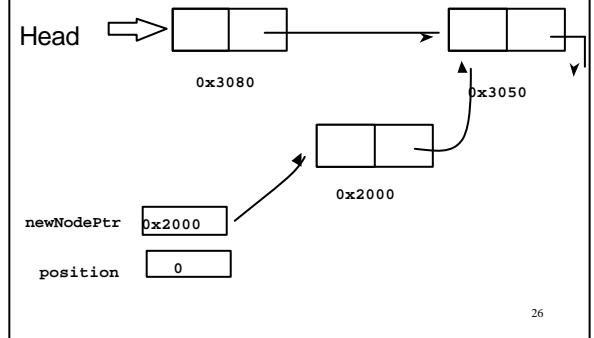


24

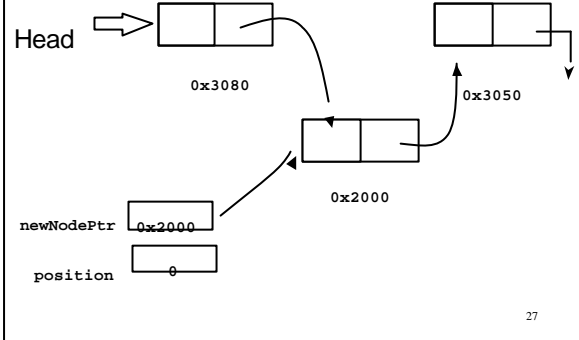
Inserting – Inside the List



Inserting – Inside the List



Inserting – Inside the List

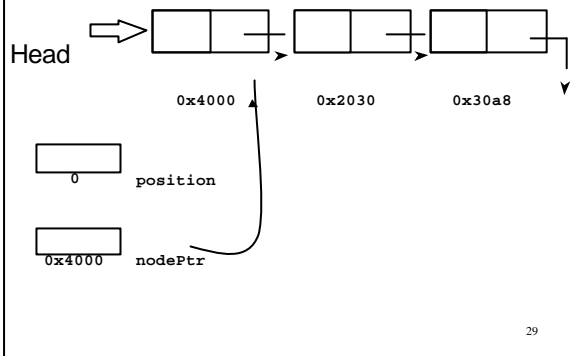


```
void insertItem(List* listPtr, float item, int position)
{
    Node* newNodePtr = makeNode(item);
    Node* nodePtr = NULL;

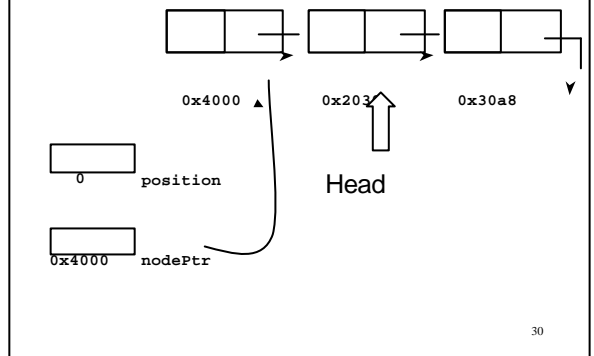
    if (position == 0)
    {
        newNodePtr->nextPtr = listPtr->headPtr;
        listPtr->headPtr = newNodePtr;
    }
    else
    {
        nodePtr = setPosition(listPtr, position-1);
        newNodePtr->nextPtr = nodePtr->nextPtr;
        nodePtr->nextPtr = newNodePtr;
    }
    listPtr->count++;
}
```

28

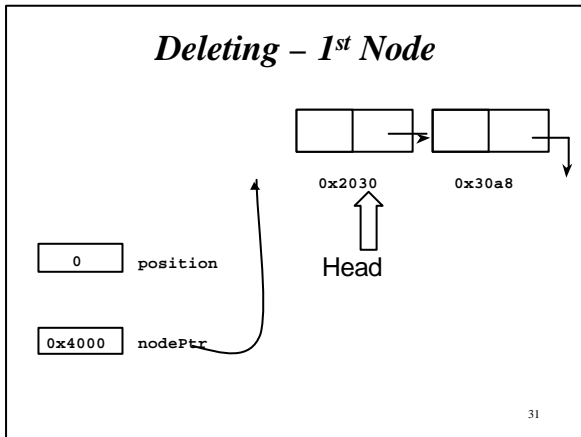
Deleting – 1st Node



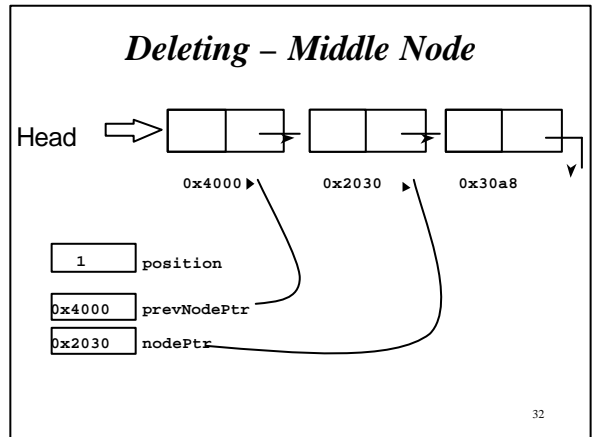
Deleting – 1st Node



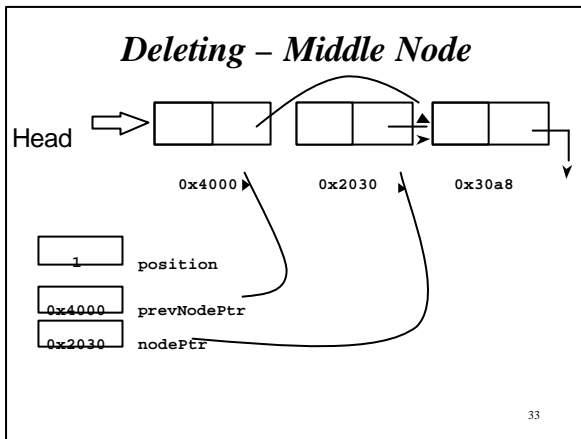
Deleting – 1st Node



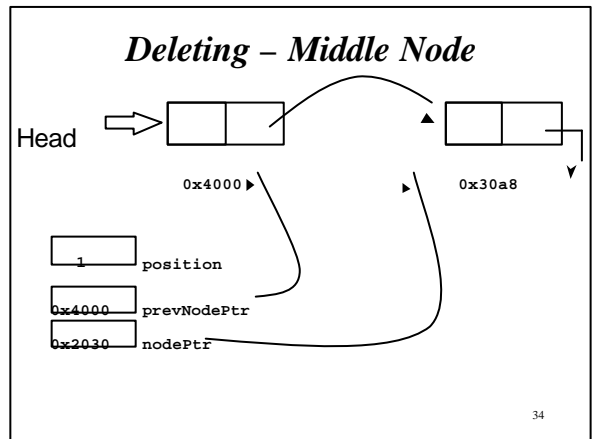
Deleting – Middle Node



Deleting – Middle Node



Deleting – Middle Node



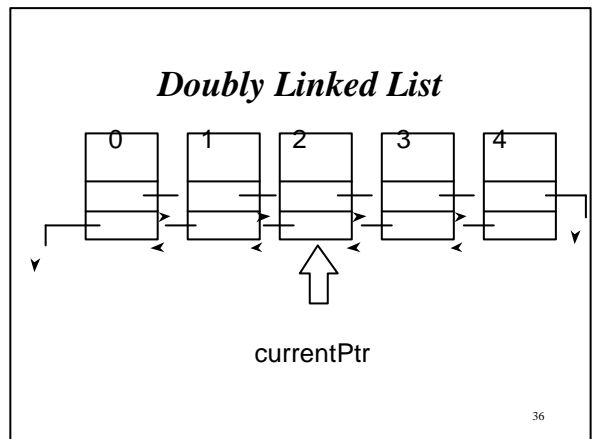
```

void deleteNode(List* listPtr, int position)
{
    Node* prevNodePtr = NULL;
    Node* nodePtr = NULL;

    if (listPtr->count > 0 && position < listPtr->count)
    {
        if (position == 0) {
            nodePtr = listPtr->headPtr;
            listPtr->headPtr = nodePtr->nextPtr;
        }
        else {
            prevNodePtr = setPosition(listPtr, position - 1);
            nodePtr = prevNodePtr->nextPtr;
            prevNodePtr->nextPtr = nodePtr->nextPtr;
        }
        listPtr->count--;
        free(nodePtr);
    }
    else {
        fprintf(stderr, "List is empty or invalid position.\n");
        exit(1);
    }
}
    
```

35

Doubly Linked List



```

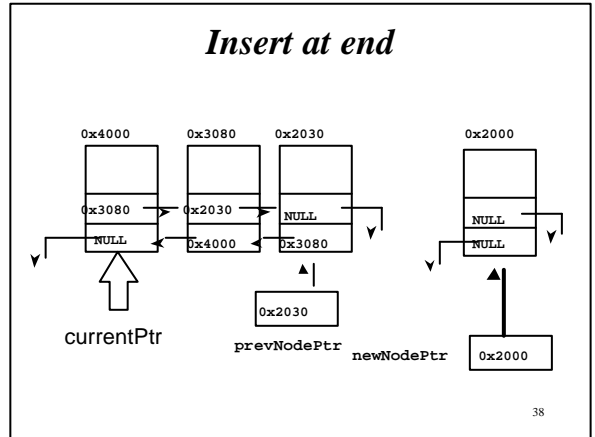
struct DoubleLinkNodeRec
{
    float          value;
    struct DoubleLinkNodeRec* nextPtr;
    struct DoubleLinkNodeRec* previousPtr;
};

typedef struct DoubleLinkNodeRec Node;

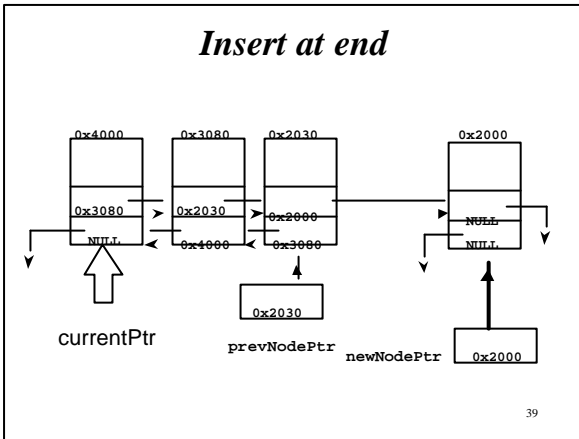
struct DoubleLinkListRec
{
    int    count;
    Node*  currentPtr;
    int    position;
};

typedef struct DoubleLinkListRec DoubleLinkList;
    
```

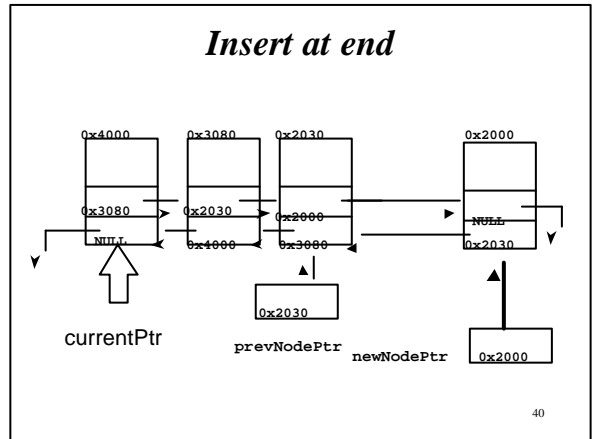
37



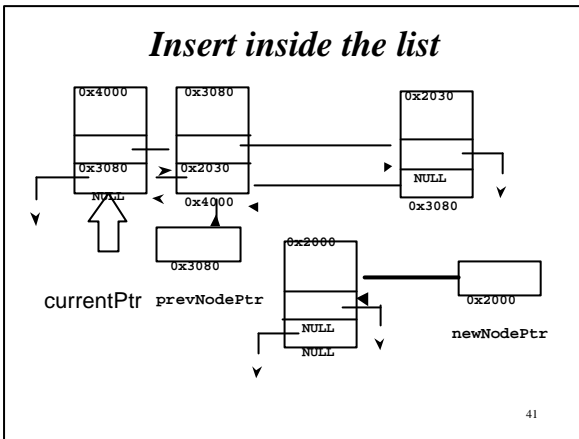
38



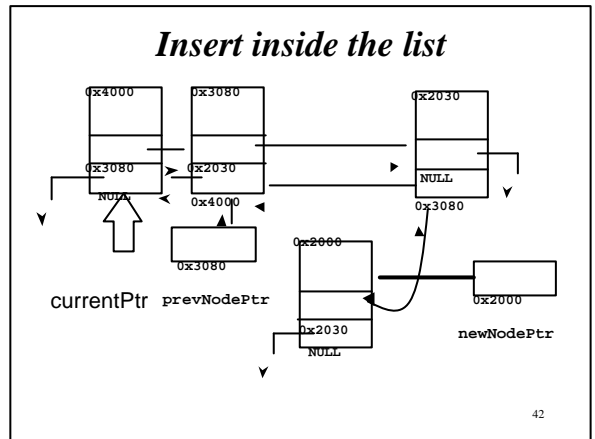
39



40

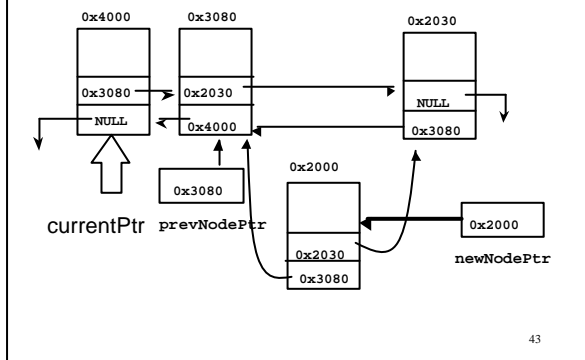


41



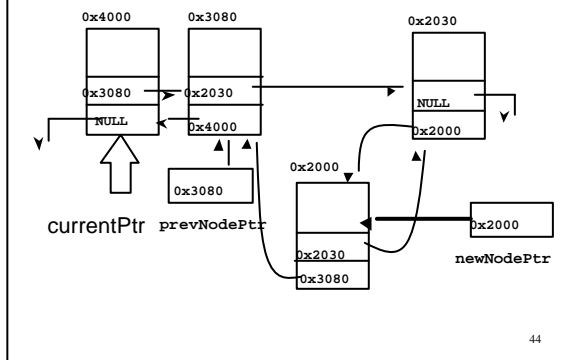
42

Insert inside the list



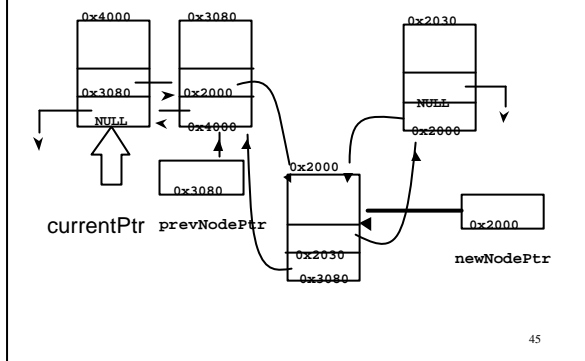
43

Insert inside the list



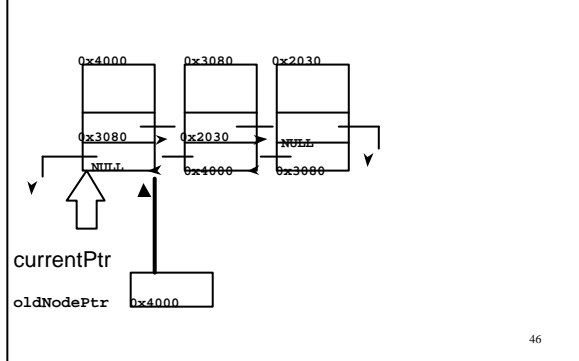
44

Insert inside the list



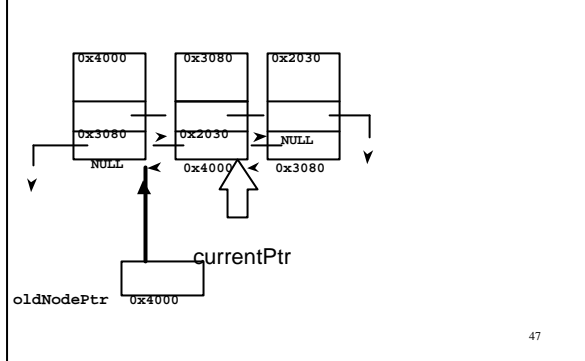
45

Delete from start



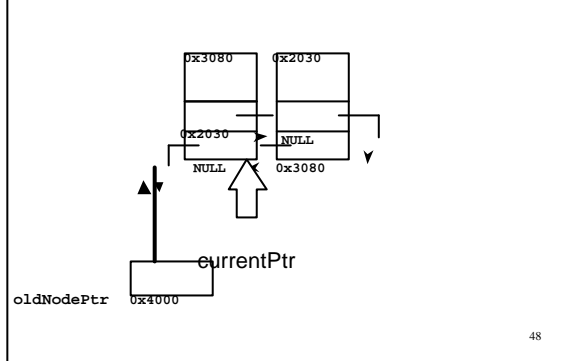
46

Delete from start



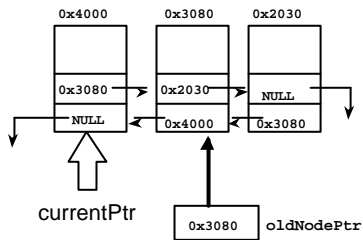
47

Delete from start



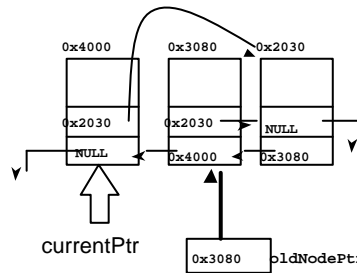
48

Delete from inside list



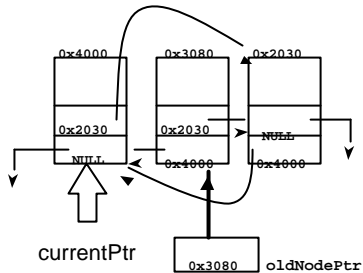
49

Delete from inside list



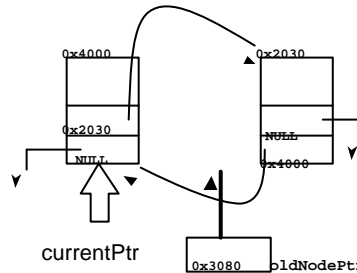
50

Delete from inside list



51

Delete from inside list



52

Revision – Recursion

- Unary
 - calls itself at most once
- Binary/N-ary
 - calls itself twice/N times
- Direct
 - the function calls itself
- Indirect
 - The function calls another function which calls the first function again.

53

Revision – Recursion

- Always
 - Converges to a base case (*always*)
 - Has a recursive definition
 - Has a base case
- Can remove recursion using a stack instead
- Disadvantages
 - May run slower
 - May use more space
- Advantages
 - Easier to prove correct
 - Easier to analyse

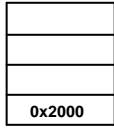
54

Recursive - Free List

nodePtr
↓

```

0x2000
/* Delete the entire list */
void FreeList(Node* nodePtr)
{
    if (nodePtr==NULL)
        return;
    FreeList(nodePtr->next);
    free(nodePtr);
}
    
```



Stack in memory

61

Revision - Binary Trees

- Parent nodes always have 2 children
- Expression Tree
 - A Binary Tree built with operands and operators.
 - Also known as a parse tree.
 - Used in compilers.
- Binary Search Tree
 - Every node entry has a **unique** key.
 - **All** the keys in the **left subtree** of a node are **less** than the key of the node.
 - **All** the keys in the **right subtree** of a node are **greater** than the key of the node

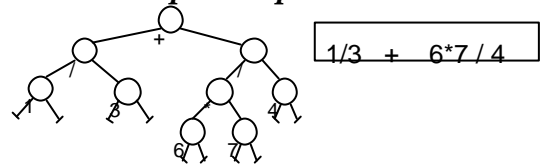
62

Revision - Binary Trees

- Traversal
 - PreOrder (Visit Left Right)
 - InOrder (Left Visit Right)
 - PostOrder (Left Right Visit)

63

Example: Expression Tree



1	/	3	+	6	*	7	/	4	Infix
1	3	/	6	7	*	4	/	+	Postfix
+	/	1	3	/	*	6	7	4	Prefix

64

Revision - Binary Search Trees

- Operations:
 - Initialise
 - Insert
 - Search
 - Delete (not needed to be known for exam)
 - Make a new node
 - Traverse (inorder, preorder, postorder)

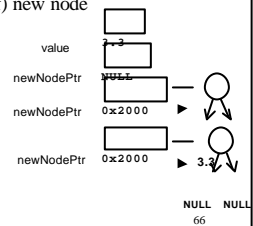
65

Make a new node

- Steps:
 - allocate memory for the new node
 - put item into the new node
 - set left and right branches to NULL
- returns: pointer to (i.e. address of) new node

```

TreeNode* makeTreeNode(float value)
{
    TreeNode* newNodePtr = NULL;
    newNodePtr = (TreeNode*)malloc(sizeof(TreeNode));
    if (newNodePtr == NULL){
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    else{
        newNodePtr->key = value;
        newNodePtr->leftPtr = NULL;
        newNodePtr->rightPtr = NULL;
    }
    return newNodePtr;
}
    
```



66

Insert Insert 0.9

```

TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
    
```

67

Insert Insert 0.9

```

TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
    
```

68

Insert Insert 0.9

```

TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
    
```

69

Insert Insert 0.9

```

TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
    
```

70

Insert Insert 0.9

```

TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
    
```

71

Search Find 0.7

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
    }
    return nodePtr;
}
    
```

72

Search Find 0.7

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
    
```

73

Search Find 0.7

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
    
```

74

Search Find 0.7

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
    
```

75

Search Find 0.5

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
    
```

76

Search Find 0.5

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
    
```

77

Search Find 0.5

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
    
```

78

Search Find 0.5

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
    
```

79

Search Find 0.5 ✗

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
    
```

80

Traversal

- Inorder traversal of a Binary Search Tree **always** gives the sorted order of the keys. (left, visit, right)


```

void printInorder(TreeNode* nodePtr){
    printInorder(nodePtr->leftPtr);
    printf(" %f", nodePtr->key);
    printInorder(nodePtr->rightPtr);
}
            
```
- Preorder (visit, left, right):


```

void printPreOrder(TreeNode* nodePtr){
    printf(" %f", nodePtr->key);
    printPreOrder(nodePtr->leftPtr);
    printPreOrder(nodePtr->rightPtr);
}
            
```
- Postorder (left, right, visit):


```

void printPostOrder(TreeNode* nodePtr){
    printPostOrder(nodePtr->leftPtr);
    printPostOrder(nodePtr->rightPtr);
    printf(" %f", nodePtr->key);
}
            
```

81

Revision - Hash Tables

- Each item has a unique key.
- Use a large array called a Hash Table.
- Use a Hash Function
 - Use prime numbers
 - Maps keys to positions in the Hash Table.
 - Be easy to calculate.
 - Use all of the key.
 - Spread the keys uniformly.
 - Use unsigned integers
- Operations:
 - Insert
 - Delete
 - Search
 - Initialise

82

Example: Hash Function for a string

```

unsigned hash(char* s){
    int i = 0;
    unsigned value = 0;
    while (s[i] != '\0'){
        value = (s[i] + 31*value) % 101;
        i++;
    }
    return value;
}
    
```

83

Linear Probing - Insert

- Apply hash function to get a position.
- Try to insert key at this position.
- Deal with collision
 - When two keys are mapped to the same position.
 - Very likely

Linear Probing - Search

- Apply hash function to get a position.
- Look at this position.
- Deal with collision
 - When two keys are mapped to the same position.
 - Very likely

Linear Probing - Delete

- Use the search function to find the item
- If found check that items after that also don't hash to the item's position
- If items after do hash to that position, move them back in the hash table and delete the item.

84

Example: Insert with Linear Probing

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick Knuth

```

module insert(hashTable, item)
{
  position = hash(item)
  initialise count to 0
  while (count < hashTableSize)
  {
    if (position in hashTable is empty)
    {
      write item at position in hashTable
      exit loop
    }
    else
    {
      step position along (wrap around)
      increment count
    }
  }
  if (count == hashTableSize) then
    the hashTable is full
}
    
```

85

Example: Search with Linear Probing

```

module search(hashTable, target)
{
  position = hash(target)
  initialise count to 0
  while (count < hashTableSize)
  {
    if (position in hashTable is empty)
      return -1;

    else if (key at position in hashTable == target)
      return position;

    step position along (wrap around)
    increment count
  }
  if (count == hashTableSize) then
    return -1;
}
    
```

86

Hashtable with Chaining

- At each position in the array you have a list:

```
List hashTable[MAXTABLE];
```

- Advantages
 - Insertions and deletions are quick & easy
 - Resizable
- Disadvantages
 - Uses more space
 - More complex to implement

87

Insert with Chaining

- Apply hash function to get a position in the array.
- Insert key into the Linked List at this position in the array.

```

void InsertChaining(Table* hashTable, float item)
{
  int posHash = hash(item)
  ListInsert (hashTable[posHash], item);
}
    
```

88

Search with Chaining

- Apply hash function to get a position in the array.
- Search the Linked List at this position in the array to see if the item is in it.

```

/* module returns NULL if not found, or the address of the
 * node if found */
Node* SearchChaining(Table* hashTable, float item){
  posHash = hash(item)
  Node* found;
  found = searchLIST (hashTable[posHash], item);
  return found;
}
    
```

89

Delete with Chaining

- Apply hash function to get a position in the array.
- Delete the item in the Linked List at this position in the array.

```

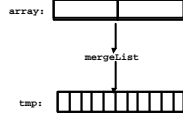
/* module uses the Linked list delete function to delete
 * an item inside that list, it does nothing if that item
 * isn't there. */
void DeleteChaining(Table* hashTable, float item){
  int posHash = hash(item)
  deleteLIST (hashTable[posHash], item);
}
    
```

90

Revision - Mergesort

- Recursively split the array in half until you have arrays of length 1
- Merge them together in sorted order
- Return the merged array

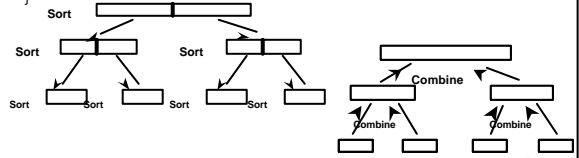
```
void mergeSort(float array[], int size){
    int* tmpArrayPtr = (int*)malloc(size*sizeof(int));
    if (tmpArrayPtr != NULL)
        mergeSortRec(array, size, tmpArrayPtr);
    else{
        fprintf(stderr, "Not enough memory to sort list.\n");
        exit(1);
    }
    free(tmpArrayPtr);
}
```



91

Revision - Mergesort

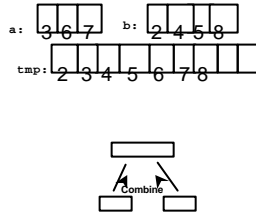
```
void mergeSortRec(float array[], int size, float tmp[]){
    int i;
    int mid = size/2;
    if (size > 1) {
        mergeSortRec(array, mid, tmp);
        mergeSortRec(array+mid, size-mid, tmp);
        mergeArrays(array, mid, array+mid, size-mid, tmp);
    }
    for (i = 0; i < size; i++)
        array[i] = tmp[i];
}
```



92

Revision - Mergesort

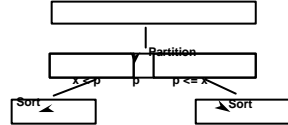
```
void mergeArrays(float a[], int aSize, float b[], int bSize, float tmp[]){
    int k, i = 0, j = 0;
    for (k = 0; k < aSize + bSize; k++)
    {
        if (i == aSize) {
            tmp[k] = b[j];
            j++;
        }
        else if (j == bSize) {
            tmp[k] = a[i];
            i++;
        }
        else if (a[i] <= b[j]) {
            tmp[k] = a[i];
            i++;
        }
        else {
            tmp[k] = b[j];
            j++;
        }
    }
}
```



93

Revision - Quicksort

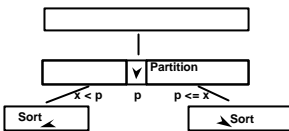
- Partition
 - Choose a pivot element
 - Swap pivot with array[0]
 - Sort remaining elements so that the ones less than the pivot are to the left of the ones that are greater than the pivot.
 - Swap the pivot back into the correct position (the rightmost less than element)
- Sort the sub-array to the left of the pivot
- Sort the sub-array to the right of the pivot



94

Revision - Quicksort

```
void quickSort(float array[], int size){
    int index;
    if (size > 1){
        index = partition(array, size);
        quickSort(array, index);
        quickSort(array+index+1, size - index - 1);
    }
}
```



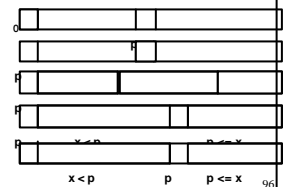
95

Revision - Quicksort - Partition

```
int partition(float array[], int size){
    int k;
    int mid = size/2;
    int index = 0;

    swap(array, array+mid);

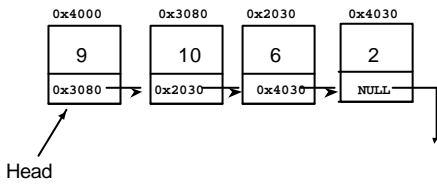
    for (k = 1; k < size; k++){
        if (list[k] < list[0]){
            index++;
            swap(array+k, array+index);
        }
    }
    swap(array, array+index);
    return index;
}
```



96

Going Further (not examinable)

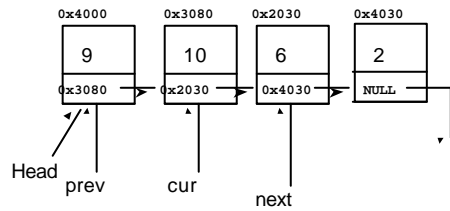
- Insertion sort on a linked list



97

Going Further (not examinable)

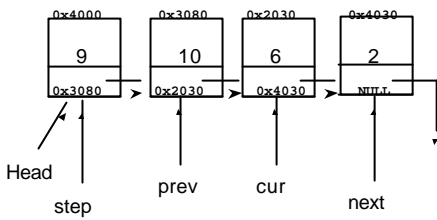
- Insertion sort on a linked list



98

Going Further (not examinable)

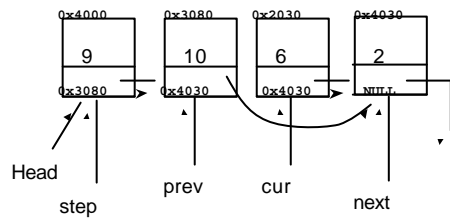
- Insertion sort on a linked list



99

Going Further (not examinable)

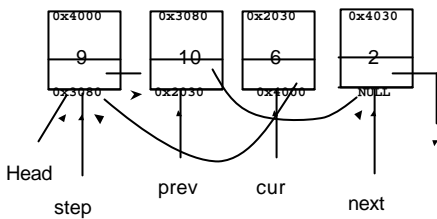
- Insertion sort on a linked list



100

Going Further (not examinable)

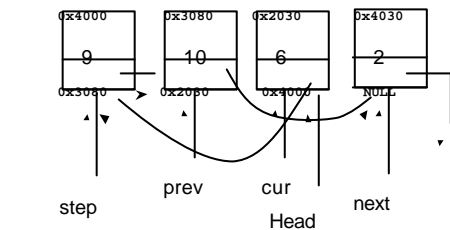
- Insertion sort on a linked list



101

Going Further (not examinable)

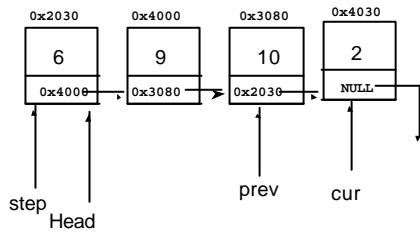
- Insertion sort on a linked list



102

Going Further (not examinable)

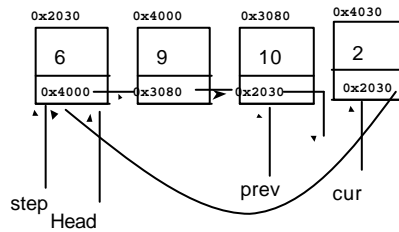
- Insertion sort on a linked list



103

Going Further (not examinable)

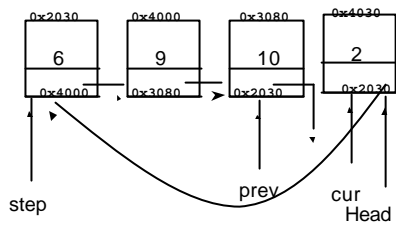
- Insertion sort on a linked list



104

Going Further (not examinable)

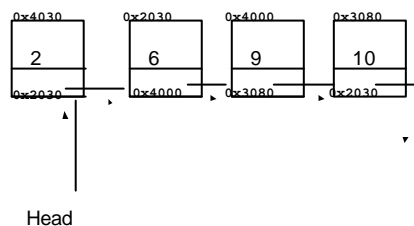
- Insertion sort on a linked list



105

Going Further (not examinable)

- Insertion sort on a linked list



106

Going Further (not examinable)

- Code for a doubly linked list follows on the next two slides.

107

```

/* Not examinable */
void insert(DoubleList* listPtr, float item, int position)
{
    Node* newNodePtr = makeNode(item);
    Node* nodePtr = NULL;

    if (position == 0)
    {
        newNodePtr->nextPtr = listPtr->headPtr;
        if (listPtr->headPtr != NULL)
            listPtr->headPtr->previousPtr = newNodePtr;
        listPtr->headPtr = newNodePtr;
    }
    else
    {
        nodePtr = setPosition(listPtr, position-1);
        newNodePtr->nextPtr = nodePtr->nextPtr;
        newNodePtr->previousPtr = nodePtr;
        nodePtr->nextPtr->previousPtr = newNodePtr;
        nodePtr->nextPtr = newNodePtr;
    }
    listPtr->count++;
}
    
```

108

```
/* Not examinable */
void deleteNode(DoubleList* listPtr, int position)
{
    Node* oldNodePtr = NULL;
    Node* nodePtr = NULL;

    if (listPtr->count > 0 && position < listPtr->count) {
        if (position == 0) {
            oldNodePtr = listPtr->headPtr;
            listPtr->headPtr = oldNodePtr->nextPtr;
            if (listPtr->headPtr != NULL)
                listPtr->headPtr->previousPtr = NULL;
        }
        else {
            nodePtr = setPosition(listPtr, position - 1);
            oldNodePtr = nodePtr->nextPtr;
            nodePtr->nextPtr = oldNodePtr->nextPtr;
            oldNodePtr->nextPtr->previousPtr = nodePtr;
        }
        listPtr->count--;
        free(oldNodePtr);
    }
    else {
        fprintf(stderr, "List is empty or invalid position.\n");
        exit(1);
    }
}
```

109