

# *Linked Lists*

CSE1303 Part A

Data Structures and Algorithms

# *Overview*

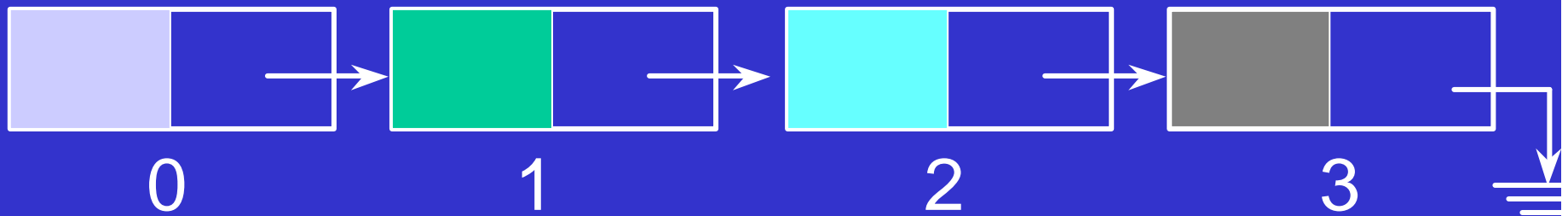
- Operations for Lists.
- Implementation of Linked Lists.
- Double Linked Lists.

# *List Operations*

- Go to a **position** in the list.
- **Insert** an item at a position in the list.
- **Delete** an item from a position in the list.
- **Retrieve** an item from a position.
- **Replace** an item at a position.
- **Traverse** a list.

# *Linked List*

Head



```
#ifndef LINKEDLISTH
#define LINKEDLISTH
#include <stdbool.h>
#include "node.h"

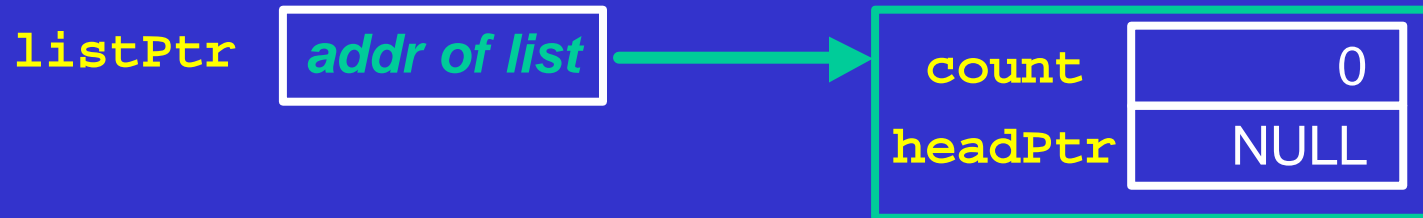
struct LinkedListRec
{
    int    count;
    Node*  headPtr;
};

typedef struct LinkedListRec List;

void initializeList(List* listPtr);
bool listEmpty(const List* listPtr);
Node* setPosition(const List* listPtr, int position);
void insertItem(List* listPtr, float item, int position);
float deleteNode(List* listPtr, int position);

#endif
```

# *Initialize List*



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

# *Set Position*

- check if **position** is within range
- start with address of **head node**
- set **count** to 0
- while count is **less** than position
  - follow link to **next node**
  - **increment** count
- return address of **current node**

```

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;
}

```

```
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;
}
```

```

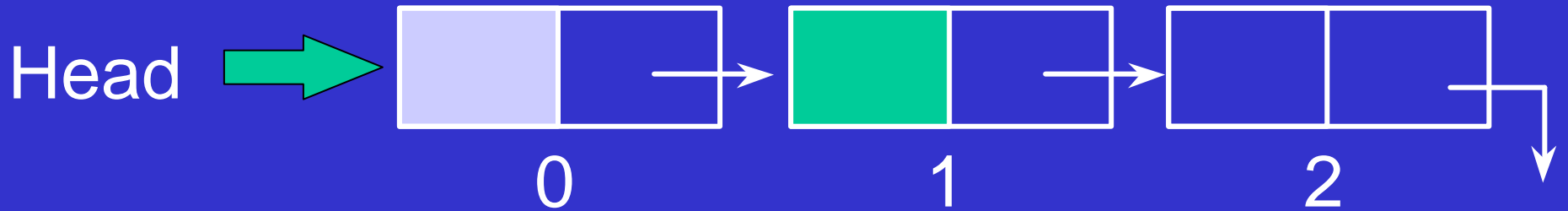
Node* setPosition(const List* listPtr, int position)
{
    int    i;
    Node*  nodePtr = listPtr->headPtr;

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

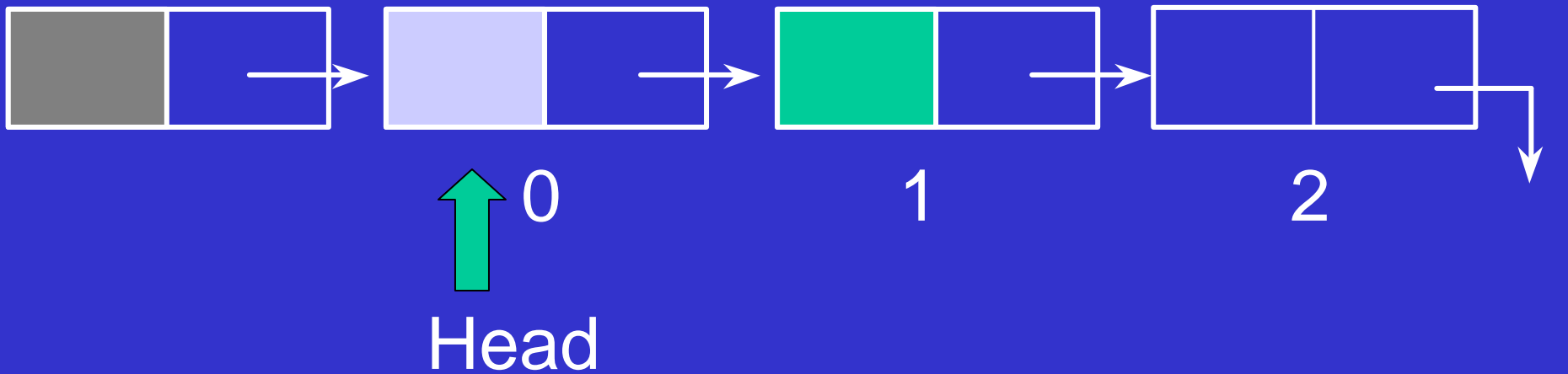
    return nodePtr;
}

```

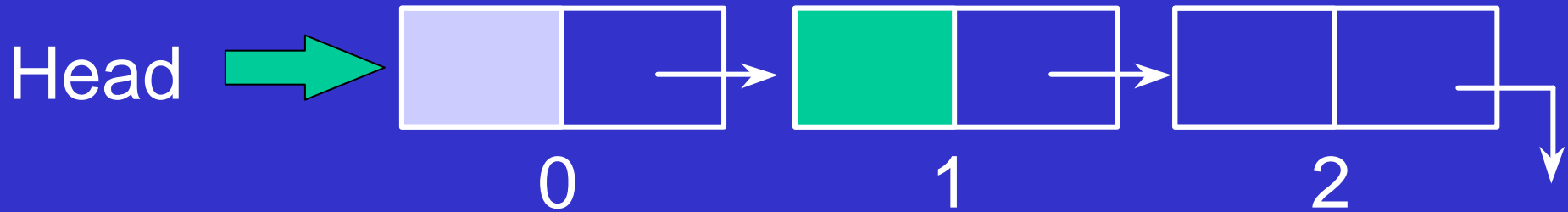
# *Insert*



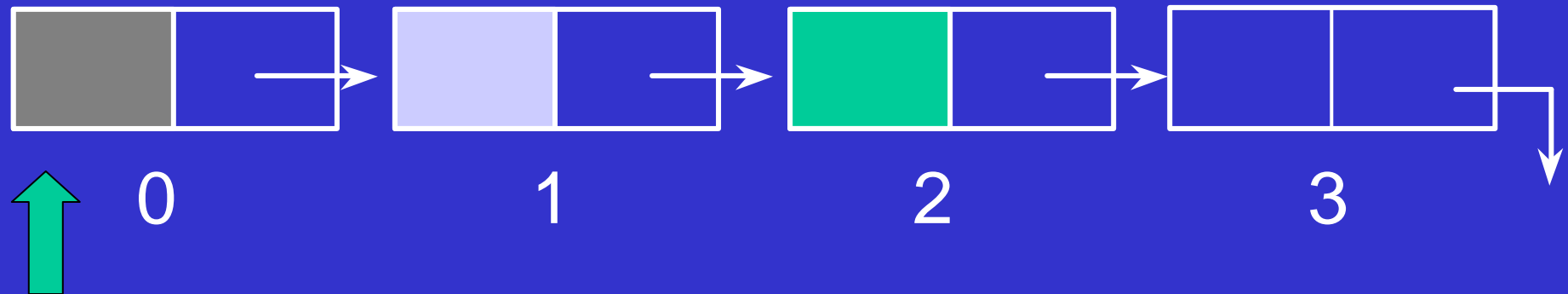
If we insert it at position 0.



# *Insert*

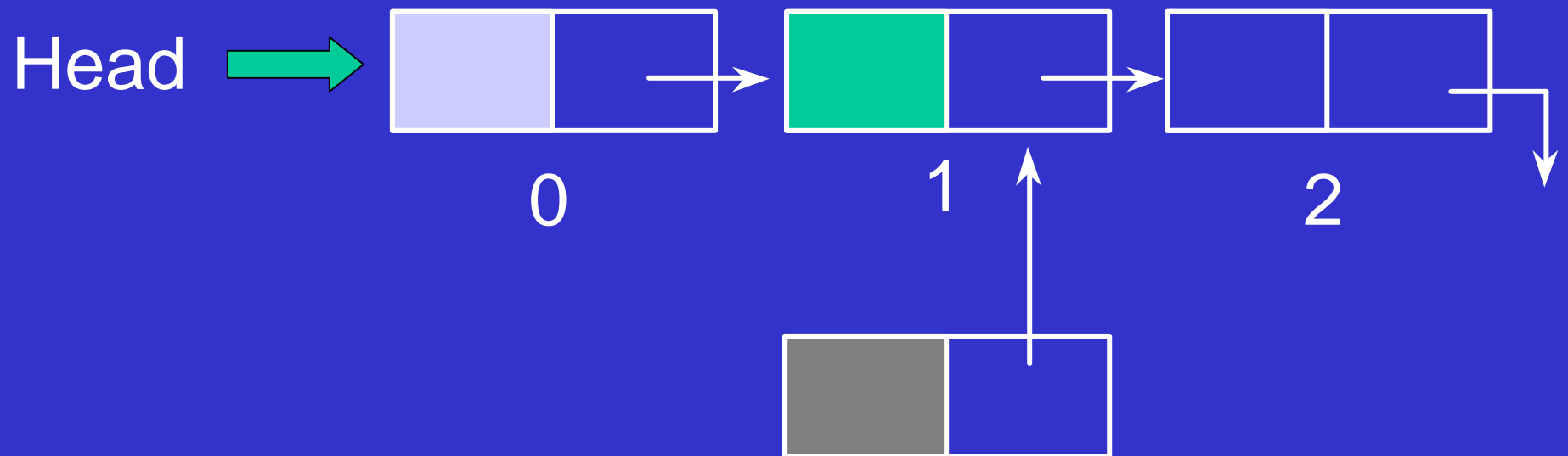


If we insert it at position 0.

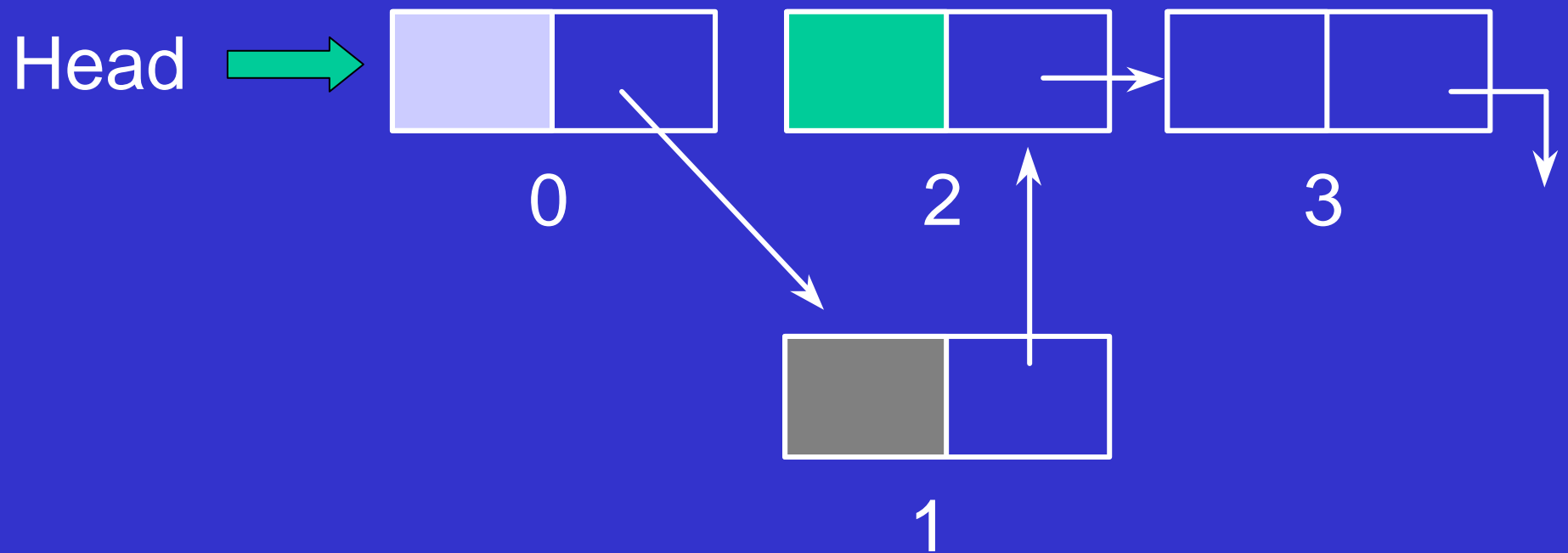


Head

Instead, suppose we insert it at position 1.



Instead, suppose we insert it at position 1.



```

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++;
}

```

```

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++;
}

```

```
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++;
}
```

```
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++;
}
```

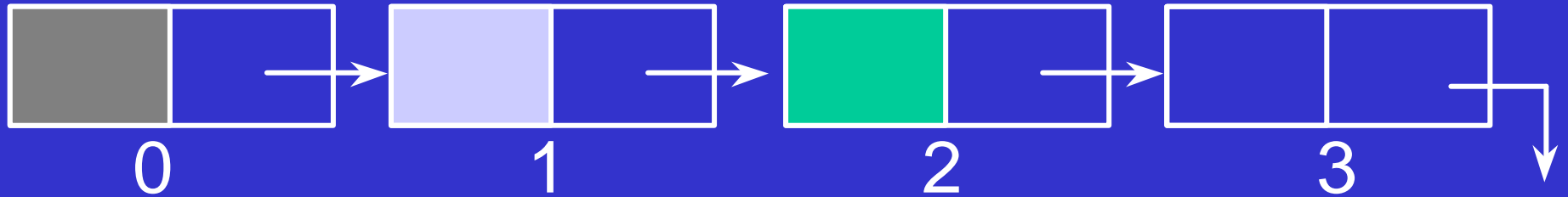
```
void insertItem(List* listPtr, float item, int position)
{
    Node* newNodePtr = makeNode(item);
    Node* nodePtr = NULL;

    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++;
}
```

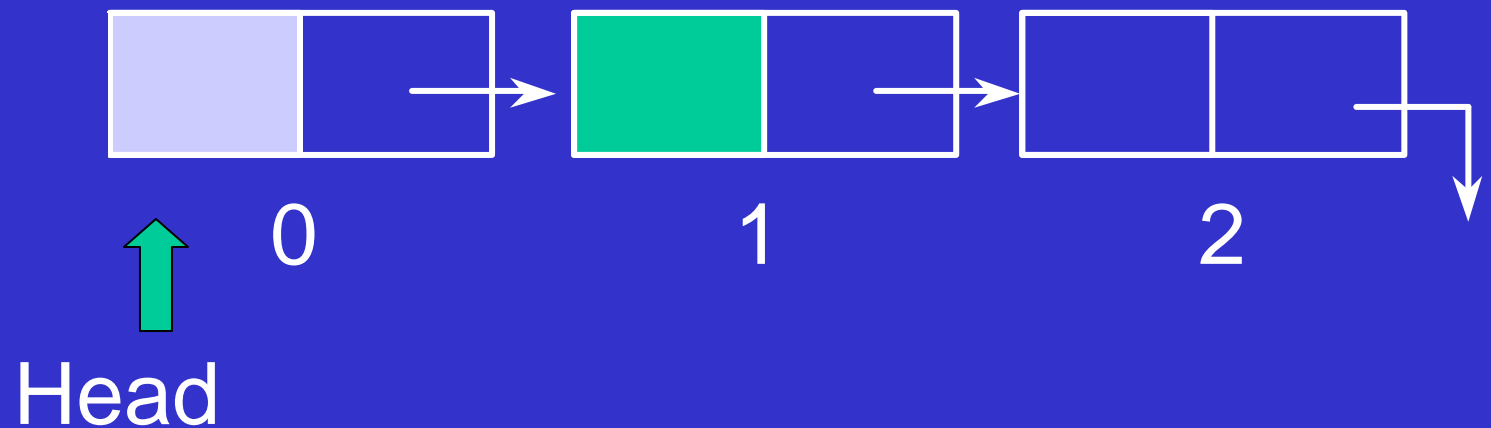
Head



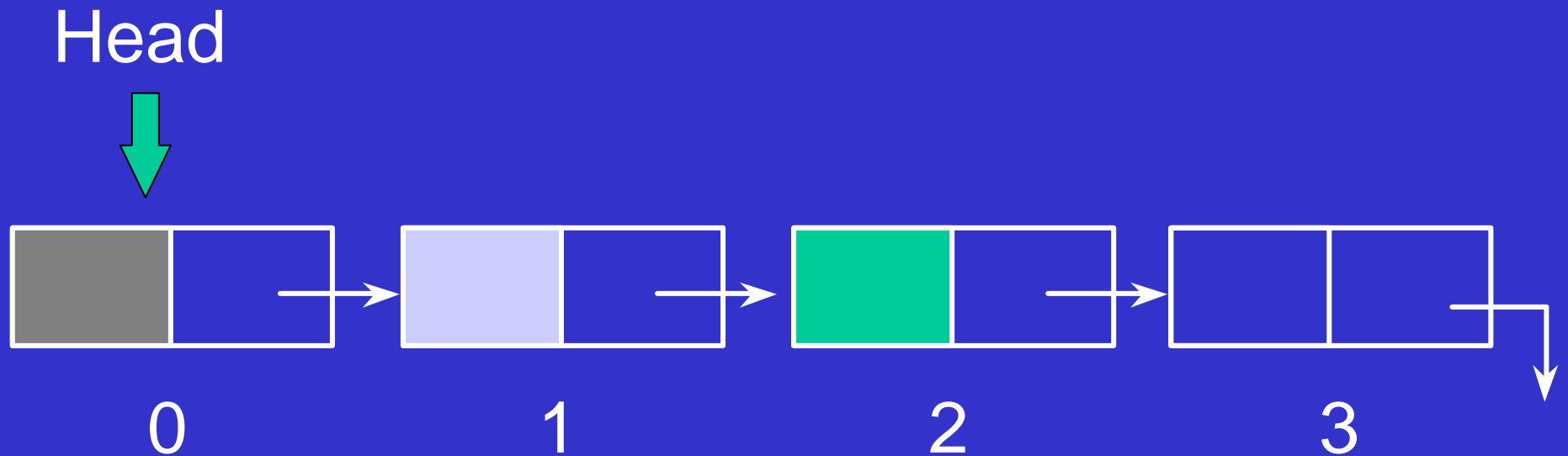
# Delete



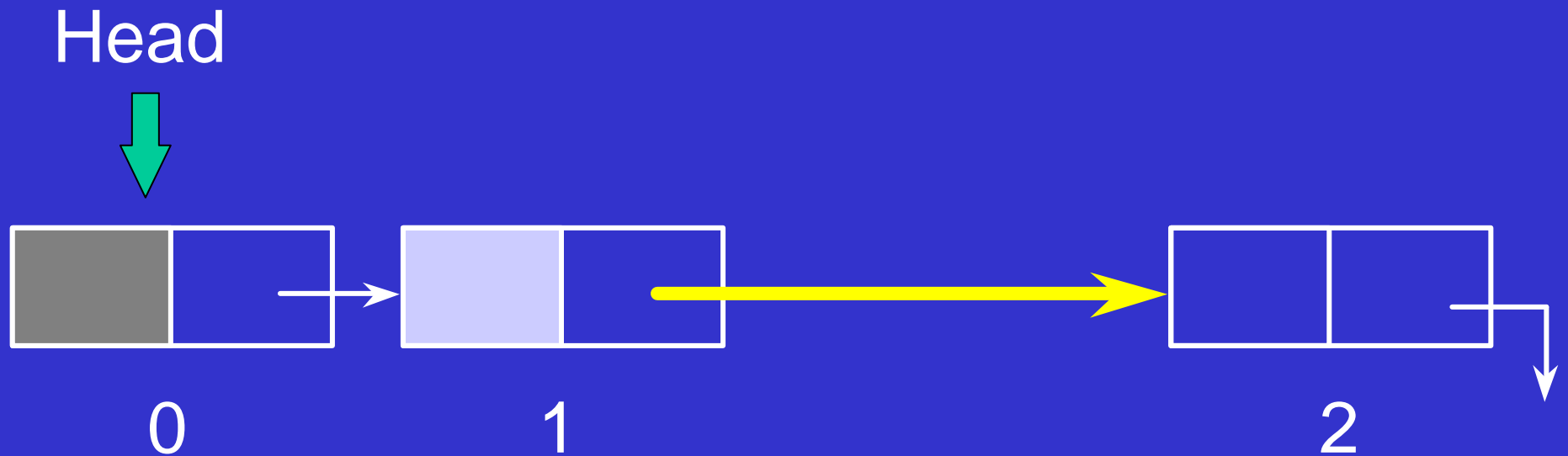
If we delete the Node at position 0.



If we delete the Node at position 2.



If we delete the Node at position 2.



```

void deleteNode(List* 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;
        }
        else {
            nodePtr = setPosition(listPtr, position - 1);
            oldNodePtr = nodePtr->nextPtr;
            nodePtr->nextPtr = oldNodePtr->nextPtr;
        }
        listPtr->count--;
        free(oldNodePtr);
    }
    else {
        fprintf(stderr, "List is empty or invalid position.\n");
        exit(1);
    }
}

```

```

void deleteNode(List* 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;
        }
        else {
            nodePtr = setPosition(listPtr, position - 1);
            oldNodePtr = nodePtr->nextPtr;
            nodePtr->nextPtr = oldNodePtr->nextPtr;
        }
        listPtr->count--;
        free(oldNodePtr);
    }
    else {
        fprintf(stderr, "List is empty or invalid position.\n");
        exit(1);
    }
}

```

```

void deleteNode(List* 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;
        }
        else {
            nodePtr = setPosition(listPtr, position - 1);
            oldNodePtr = nodePtr->nextPtr;
            nodePtr->nextPtr = oldNodePtr->nextPtr;
        }
        listPtr->count--;
        free(oldNodePtr);
    }
    else {
        fprintf(stderr, "List is empty or invalid position.\n");
        exit(1);
    }
}

```

```

void deleteNode(List* 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;
        }
        else {
            nodePtr = setPosition(listPtr, position - 1);
            oldNodePtr = nodePtr->nextPtr;
            nodePtr->nextPtr = oldNodePtr->nextPtr;
        }
        listPtr->count--;
        free(oldNodePtr);
    }
    else {
        fprintf(stderr, "List is empty or invalid position.\n");
        exit(1);
    }
}

```

```

void deleteNode(List* 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;
        }
        else {
            nodePtr = setPosition(listPtr, position - 1);
            oldNodePtr = nodePtr->nextPtr;
            nodePtr->nextPtr = oldNodePtr->nextPtr;
        }
        listPtr->count--;
        free(oldNodePtr);
    }
    else {
        fprintf(stderr, "List is empty or invalid position.\n");
        exit(1);
    }
}

```

# *Comparision*

## **Linked Storage**

- Unknown list size.
- Flexibility is needed.

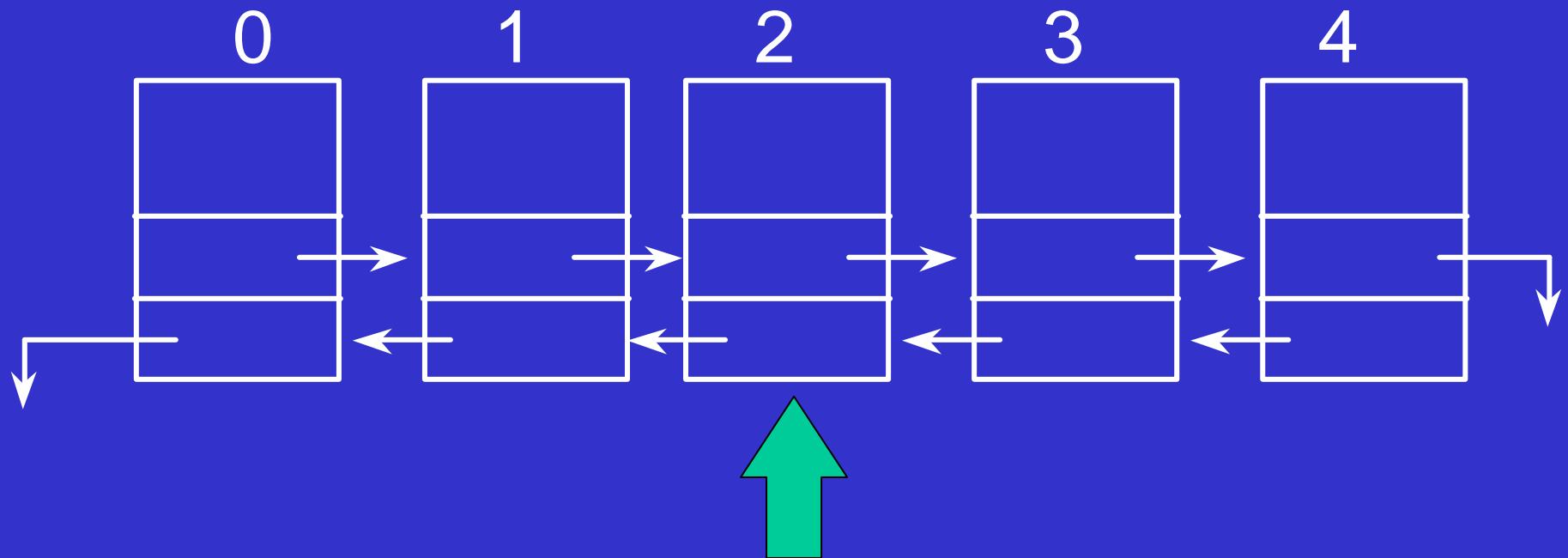
## **Contiguous Storage**

- Known list size.
- Few insertions and deletions are made within the list.
- Random access

# *Double Linked List Operations*

- Go to a position in the list.
- Insert an item in a position in the list.
- Delete an item from a position in the list.
- Retrieve an item from a position.
- Replace an item at a position.
- Traverse a list, *in both directions*.

# *Double Linked List*



Current

```
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;
```

# *Revision*

- Linked List.
- Benefits of different implementations.
- Double Linked List.

# *Preparation*

- Read Chapter 6 in Kruse et al.