Lists in C++

Dr. Robert Amelard

ramelard@uwaterloo.ca

Objectives

- Introducing the List ADT
- Nodes and Linked Lists
- Inserting a New Node into a List
- Deleting the Last Node from a List
- Printing and Searching the List Contents
- Circularly Linked Lists
- Doubly Linked Lists
- Sequential List Implementation
- Linked List Implementation

List ADT /1

- **List ADT Operations:**
  - **Insert:** inserts a value into the list at a specified position
  - **Delete:** removes data from the list at a given position
  - **Select:** returns the value stored at a given position
  - **Replace:** replaces the value stored at a given position
  - **Size:** returns the number of elements in the list

- **`void insert(DataType value, int position):`**
  1. Inserts **`value`** into the list at the given **`position`**
  2. After inserting the value, all of the data at **`position`** and after it is shifted towards the list end by one

List ADT /2

- **`void delete(int position):`**
  1. Removes data from the list at the given **`position`**
  2. On deletion, all of the existing data after **`position`** is moved towards the list front by one

- **`DataType select(int position):`**
  - Returns the value stored at the given **`position`**

- **`void replace(int position, DataType value):`**
  - Replaces with **`value`** the data stored at **`position`**

- **`int size():`**
  - Returns the number of elements in the list

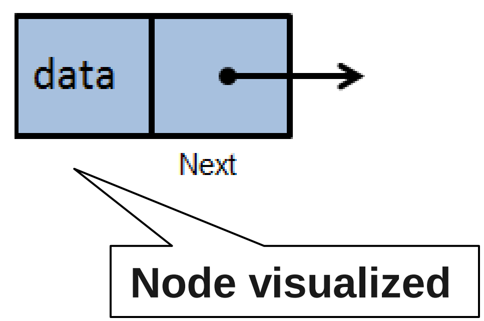
Nodes and Linked Lists /1

- **How can we use pointers to store and manipulate data in a program?**
  - Pointers can be used to address not only primitive types, but also structures and objects
  - We can create objects that store relevant information, and then chain them together into a matching structure

- **Node:**
  - A data structure that contains both a data item and a pointer to the next node in the chain/sequence

```
class Node {
        int data;
        Node* next;
        …
};
```

Node in C++

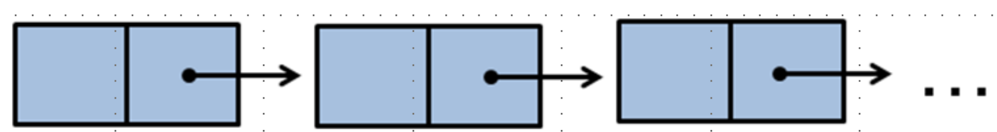


Nodes and Linked Lists /2

- **Linked List:**
  - An ordered chain/sequence of nodes, where the pointer of each node points to the next node in the list
    (i.e., each node is linked to its successor)
  - By adding or removing links from the data sequence, the list structure can accommodate the needs of an algorithm
  - Nodes can be added to the chain to expand the storage space of the list
  - Similarly, nodes can be removed when not needed

```
class LinkedList {
        Node* head;
        …
};
```

Linked List in C++



Head node

Linked List visualized

Nodes and Linked Lists /3

- **Node and `LinkedList` sample implementation:**

```
// node.hpp
class Node {
public:
    Node(int value);
    ~Node();
    Node* getNext();
    void setNext(Node* newNext);
    int getData();
    void setData(int newData);
private:
    int data;  // this node stores integer data
    Node* next;
};
```

```
// linkedlist.hpp
#include "node.hpp"
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    Node* getHeadNode();
    void setHeadNode(Node* newHead);
private:
    Node* head;
};
```

Nodes and Linked Lists /4

- **Node and `LinkedList` sample implementatio**

```
// node.hpp
class Node {
public:
    Node(int value);
    ~Node();
    Node* getNext();
    void setNext(Node* newNext);
    int getData();
    void setData(int newData);
private:
    int data;  // this node stores integer data
    Node* next;
};
```

```
// node.cpp
#include "node.hpp"
Node::Node(int value)
{

}

Node::~Node()
{

}

Node* Node::getNext()
{

}

void setNext(Node* newNext)
{

}

int Node::getData()
{

}

void Node::setData(int newData)
{

}
```

Nodes and Linked Lists /5

```
// linkedlist.hpp
#include "node.hpp"
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    Node* getHeadNode();
    void setHeadNode(Node* newHead);
private:
    Node* head;
};
```

- **Node and `LinkedList` sample implementation:**

```
// linkedlist.cpp
#include "linkedlist.hpp"
LinkedList::LinkedList()
{

}

LinkedList::~LinkedList()
{

}
```

```
Node* LinkedList::getHeadNode()
{

}

void LinkedList::setHeadNode(Node* newHead)
{

}
```

Nodes and Linked Lists /6

- **The list itself only references the first (head) node**
  - To access the remainder of the data stored in the list, we can iterate through each node using the next pointer
  - For example, `head->next->next` points to the third node in the list
    - Discussion question: what happens if you do `head->next->next->next->next->next` on a 5-item list?

- **The structure of a linked list is simple and flexible**
  - It can afford us an opportunity to practice the use of pointers in C++
  - To that end, we will implement three key operations:
    (1) **inserting** a node into a list, (2) **deleting** the last node, and
    (3) **printing** and searching inside the list contents

Inserting a New Node into a List /1

- **Let us approach this problem using bottom-up problem solving, using the following steps:**
  - (Step1) Address the smallest problem sizes – also known as **base cases** – and encode those in C/C++
  - (Step2) Address the **general cases**, and encode those in C/C++ too
  - (Step3) Group solutions for Step1 and Step2 into a function, develop test cases to check correctness, and refine the code until it passes all the required tests

Inserting a New Node into a List /2

- **(Step1) Address the smallest problem sizes (i.e., base cases), and encode them in C/C++**
  - Consider an empty list, with no nodes and no data
  - If the list is empty, head is a NULL pointer
  - Therefore, to insert a new node into an empty list, we need to create an appropriate new node, and set the head pointer to point to it
  - This can be reflected in code as follows:

```
LinkedList* list = new LinkedList(); // create a new list

Node* newNode = new Node(3); // create a new node

list->setHeadNode(newNode);  // set the head pointer to
                             // point to the new node
```

What happens behind the scenes?

Inserting a New Node into a List /3

- **(Step2) Address the general cases, and encode those in C/C++ too**
  - Consider a list that is not empty
  - In that case, the new node should go at the end of the list
  - To find the end, we follow the sequence of nodes until we encounter NULL, and then insert the new node there
  - This can be reflected in code as follows:

```
Node* end = list->getHeadNode(); // initialize end pointer
while (end->getNext() != NULL) // iterate up to NULL value
       end = end->getNext();

Node* newNode = new Node(17); // create a new node
end->setNext(newNode); // insert the new node at the end
```

Inserting a New Node into a List /4

- **(Step3) Group solutions for Step1 and Step2 into a function, and test the code to ensure correctness**

Note functional call
(not a class method)

```
void InsertNode (LinkedList* list, Node* newNode)  {
       if (list == NULL || newNode == NULL) return;
              // check for NULL values

       if (list->getHeadNode() == NULL) { // base case: empty list
              list->setHeadNode(newNode); // set head to
                            // point to the new node

       } else { // general case: non-empty list
              Node* end = list->getHeadNode(); // init end ptr
              while (end->getNext() != NULL) // iterate until NULL
                     end = end->getNext();
              end->setNext(newNode); // insert at the end
       }
}
```

Inserting a New Node into a List /5

- **(Step3) Group solutions for Step1 and Step2 into a function, and test the code to ensure correctness**
  - What do we need to test? To start, we should test:
    - A NULL list
    - A NULL new node
    - A non-NULL list with no data (i.e., head == NULL)
    - A non-NULL list with one node (i.e., head != NULL)
    - A non-NULL new node
    - Several (at least three) lists of random size, where each list has more than one node
    - Circular references where the same node is inserted many times
    - The limits of list storage (i.e., very large lists), and see how many nodes can be inserted before the program crashes

Deleting the Last Node /1

- **(Step1) Address the smallest problem sizes (i.e., base cases), and encode them in C/C++**
  - Consider an empty list
  - If the list is empty, there is nothing to delete and there is nothing to do (just exit the function, if needed)

There is nothing to do in this case!

Deleting the Last Node /2

- **(Step1) Address the smallest problem sizes (i.e., base cases), and encode them in C/C++**
  - Consider a list with one node
  - If the list has one (head) node, then the head node is the one to be deleted
  - Therefore, to delete the last node, we need to free its memory, and set the head pointer to NULL
  - This can be reflected in code as follows:

```
delete list->getHeadNode();   // free memory for the head node

list->setHeadNode(NULL);      // set the head node to NULL
                              // (avoid dangling pointer)
```

Deleting the Last Node /3

- **(Step2) Address the general cases, and encode those in C/C++ too**
  - Consider a list that has more than one node
  - In that case, we need to find the last node and delete it
  - To find the last node, we follow the sequence of nodes until the second-last node, free the memory for the last node, and set the second-last node to point to NULL
  - This can be reflected in code as follows:

```
Node* last = list->getHeadNode(); // initialize last pointer
while (last->getNext()->getNext() != NULL)
                      // iterate until the second last node
        last = last->getNext();

delete last->getNext(); // free memory for the last node
last->setNext(NULL); // set the second-last to point to NULL
```

Deleting the Last Node /4

- **(Step3) Group solutions for Step1 and Step2 into a function, and test the code to ensure correctness**

```
void DeleteLastNode(LinkedList* list) {
        if (list == NULL) return; // check for NULL values
        if (list->getHeadNode() == NULL) { // base case: empty list
                return; // nothing to do

        } else if (list->getHeadNode()->getNext() == NULL) {
                                                // base case: list with one node
                delete list->getHeadNode();
                list->setHeadNode(NULL);

        } else {         // general case: list with more than one node
                Node* last = list->getHeadNode(); // initialize last pointer
                while (last->getNext()->getNext() != NULL)
                                        // iterate until the second last node
                        last = last->getNext();

                delete last->getNext(); // free memory for the last node
                last->setNext(NULL); // set the second-last to point to NULL
        }
}
```

Deleting the Last Node /5

- **(Step3) Group solutions for Step1 and Step2 into a function, and test the code to ensure correctness**
  - What do we need to test? To start, we should test:
    - A NULL list
    - A non-NULL list with no data (i.e., head == NULL)
    - A non-NULL list with one node (i.e., head != NULL)
    - Several (at least three) lists of random size, where each list has more than one node
    - Circular references where the same node is inserted many times

Printing and Searching the List Contents /1

- **(Step1) Address the smallest problem sizes (i.e., base cases), and encode them in C/C++**
  - Consider an empty list
  - If the list is empty, there is no data to print
  - In that case, just print the header and footer
  - This can be reflected in code as follows:

```
cout << "("; // print the header data
// nothing else to do ☺
cout << ")\n"; // print the footer data
```

Printing and Searching the List Contents /2

- **(Step2) Address the general cases, and encode those in C/C++ too**
  - Consider a list that is not empty
  - To print the list contents, we follow the sequence of nodes until we encounter NULL, and print the data inside nodes at each step
  - This can be reflected in code as follows:

```
cout << "("; // print the header data
Node* last = list->getHeadNode(); // initialize last pointer
while (last != NULL) { // iterate until NULL
        cout << "[" << last->getData() << "]"; // print node
        last = last->getNext();
        if (last != NULL) // print -> symbol between nodes
                cout << "->";
}
cout << ")\n"; // print the footer data
```

Printing and Searching the List Contents /3

- **(Step3) Group solutions for Step1 and Step2 into a function, and test the code to ensure correctness**

```
void PrintNodes(LinkedList* list) {
        if (list == NULL) return; // check for NULL values

        cout << "("; // print the header data
        Node* last = list->getHeadNode(); // initialize last ptr
        while (last != NULL) { // iterate until NULL
                cout << "[" << last->getData() << "]"; // print node
                last = last->getNext();

                if (last != NULL) // print -> symbol between nodes
                        cout << "->";
        }
        cout << ")\n"; // print the footer data
}
```

Printing and Searching the List Contents /4

- **(Step1/2/3) The solution is based on the solution for printing the list contents**
  - We follow the sequence of nodes from the head node
  - If the desired value is found, we return true
  - If the desired value is not found and NULL is reached instead, we return false

```
bool FindValue(LinkedList* list, int value) {
        if (list == NULL) return false; // check for NULL values

        Node* last = list->getHeadNode(); // initialize last pointer
        while (last != NULL) { // iterate until NULL
                if (last->getData() == value) // return true if found
                        return true;

                last = last->getNext();
        }
        return false; // return false if value is not found
}
```

Printing and Searching the List Contents /5

- **(Step3) Group solutions for Step1 and Step2 into a function, and test the code to ensure correctness**
  - What do we need to test? To start, we should test:
    - A NULL list
    - A non-NULL list with no data (i.e., head == NULL)
    - A non-NULL list with one node (i.e., head != NULL)
    - Several (at least three) lists of random size, where each list has more than one node
    - Circular references where the same node is inserted many times

Also see: https://en.wikipedia.org/wiki/Doubly_linked_list

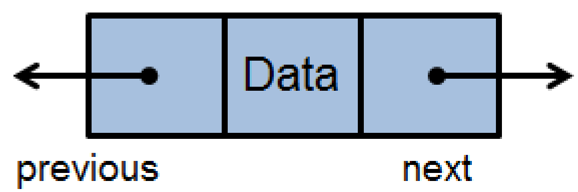
Doubly Linked Lists /1

- **Doubly Linked List:**
  - Consists of two-way (bidirectional) nodes
  - A two-way node contains two pointers: one pointer to the next, and one pointer to the previous node in the list
  - Hence, this list allows us to move both forwards and backwards through its nodes

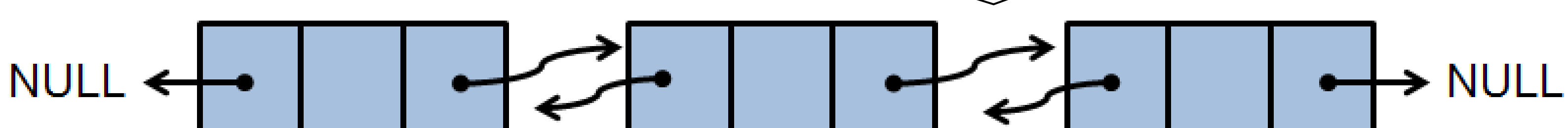
```
class DLLNode{
        int data;
        DLLNode* next;
        DLLNode* prev;
…
};
```

Two-way node in C++

Two-way node visualized



Doubly linked list visualized



Also see: https://en.wikipedia.org/wiki/Doubly_linked_list

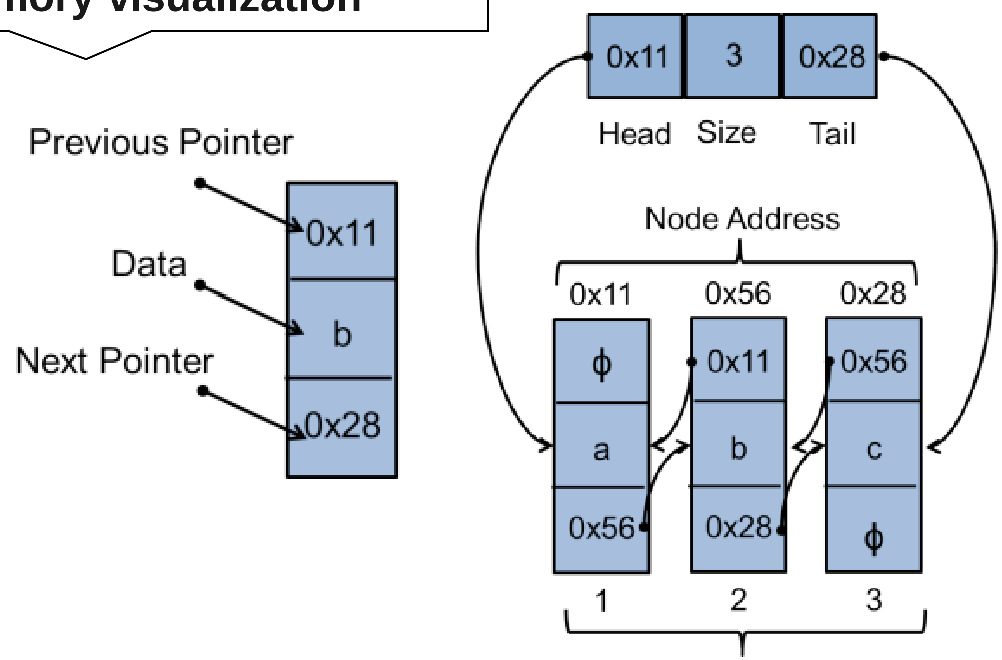
Doubly Linked Lists /2

- **Doubly Linked List:**
  - Consists of two-way (bidirectional) nodes
  - A two-way node contains two pointers: one pointer to the next, and one pointer to the previous node in the list
  - Hence, this list allows us to move both forwards and backwards through its nodes

Memory visualization

```
class DLLNode{
        int data;
        DLLNode* next;
        DLLNode* prev;
…
};
```

Two-way node in C++



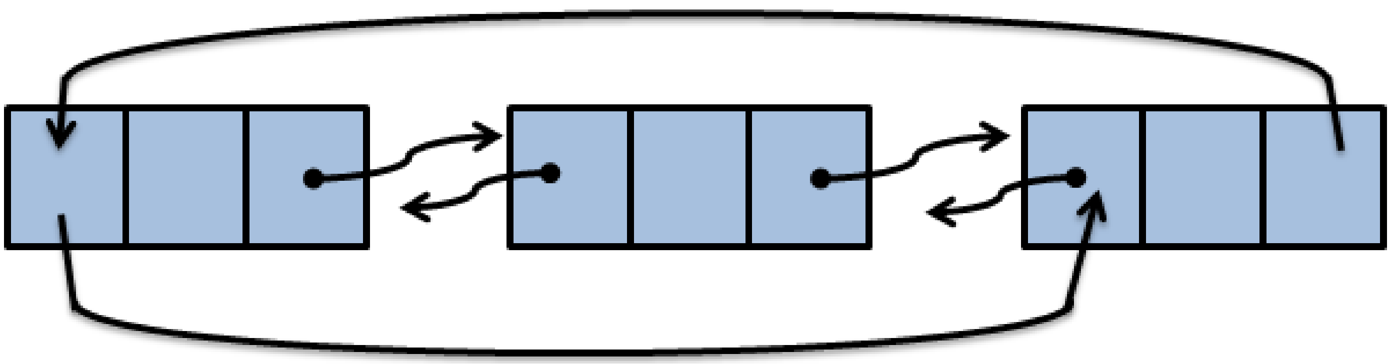
Singly vs Doubly Linked List

- Discussion: pros/cons of Singly vs Doubly Linked List?
  - `delete(Node* node)`
  - Memory & Computational requirements

Circularly Linked Lists

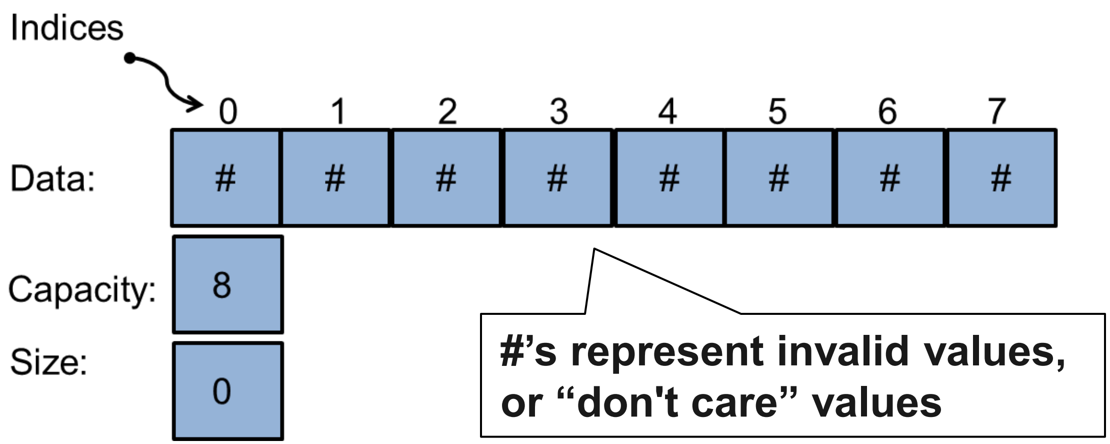
- **Circularly Linked List:**
  - The last node loops back and points to the first node
  - That is, the last node does not point to NULL as it does in a linearly linked list
  - In a doubly linked and circularly linked list, the first node also loops back and points to the last node

- **Example: doubly linked and circularly linked list**



Sequential List Implementation /1

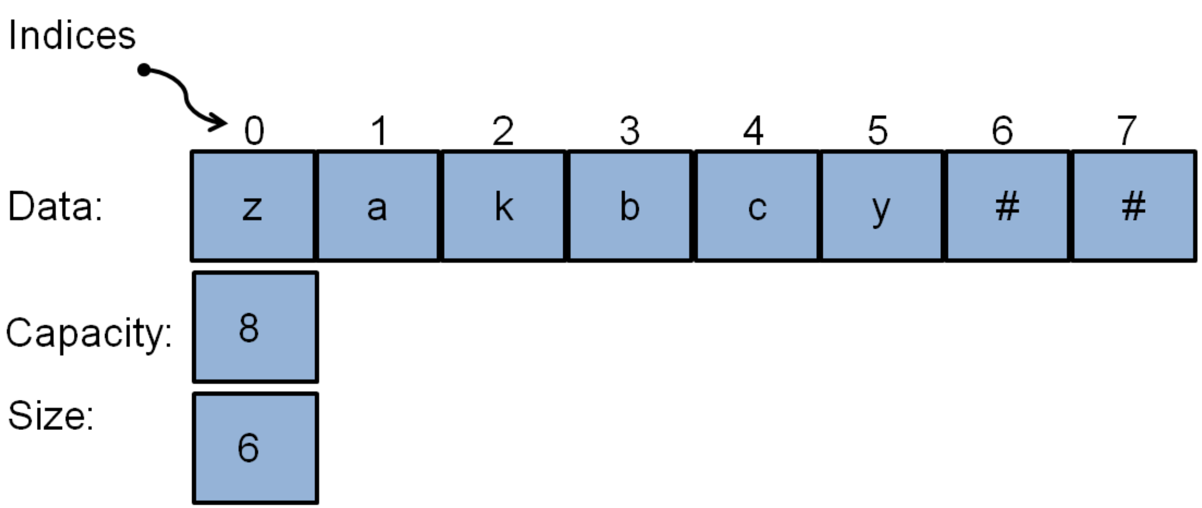
- **Sequential List Implementation:**
  - Implementing List ADT using a sequential list requires that a continuous block of memory be allocated
  - We will assume dynamic memory allocation at runtime, and the block of memory cannot be resized once created (this will be addressed later)
  - This implementation will also use the following variables:
    - **Data** – a contiguous memory location
    - **Capacity** – the maximum number of elements that data can hold
    - **Size** – the number of inserted elements in the list



Sequential List Implementation /2

- **`void insert(DataType value, int position):`**
  1. Inserts **`value`** into the list at the given **`position`**
  2. After inserting the value, all of the data at **`position`** and after it is shifted towards the list end by one

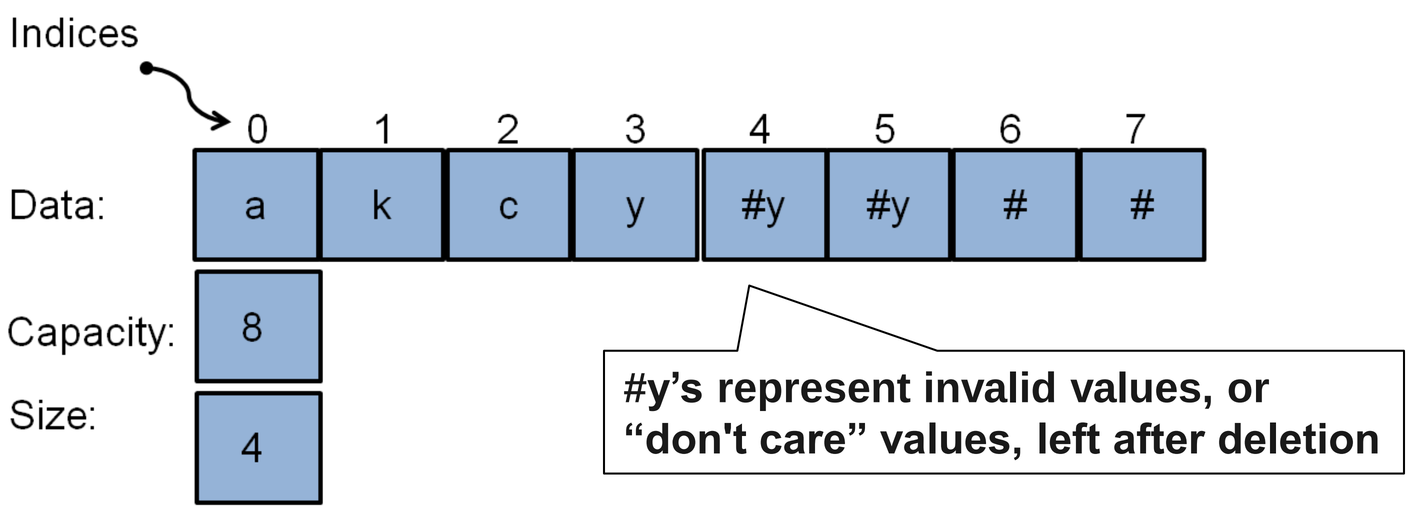
- **Example:**
  - insert('c',0) => insert(‘b',0) => insert(‘a',0) => insert(‘k',1) => insert(‘z',0) => insert(‘y',5) => insert(‘m',7) [rejected]



Sequential List Implementation /3

- **`void delete(int position):`**
  1. Removes data from the list at the given **`position`**
  2. On deletion, all of the existing data after **`position`** is moved towards the list front by one

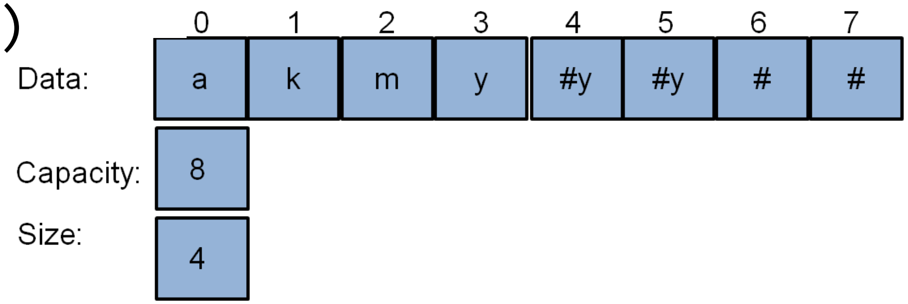
- **Example:**
  - delete(3) => delete(0)



Sequential List Implementation /4

- **`DataType select(int position):`**
  - Returns the value stored at the given **`position`**
  - Implemented as simple array lookup (e.g., data[5]) ← Fast!

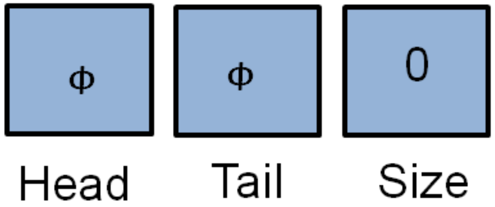
- **`void replace(int position, DataType value):`**
  - Replaces with **`value`** the data stored at **`position`**
  - Implemented as simple array replacement
  - Example: `replace(2,‘m’)`



- **`int size():`**
  - Returns the number of elements in the list
  - Implemented by returning **`size`**

Linked List Implementation /1

- **Linked List Implementation:**
  - A linked list implementation does not require that continuous memory be allocated for the list
  - The implementation will be based on the linked-list operations described earlier in the notes, but we will utilize a doubly linked list instead
  - This implementation will also use the following variables:
    - Head – pointer to the start of the list
    - Tail – pointer to the end of the list
    - Size – the number of inserted elements in the list

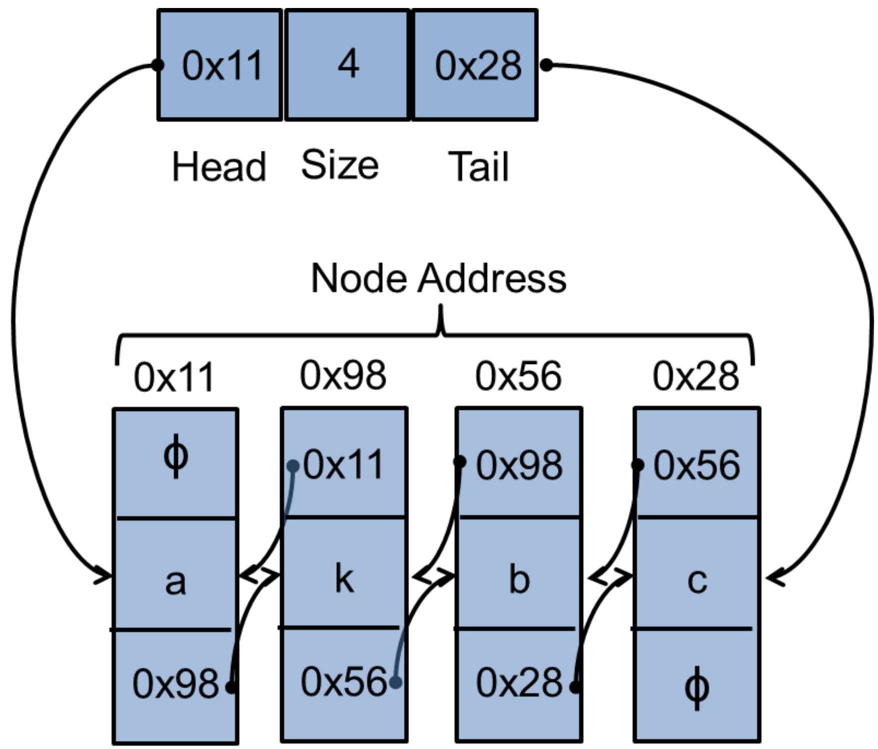


Φ’s represent NULL values

Linked List Implementation /2

- `void insert(DataType value, int position):`
  1. Inserts **`value`** into the list at the given **`position`**
  2. After inserting the value, all of the data at **`position`** and after it is shifted towards the list end by one

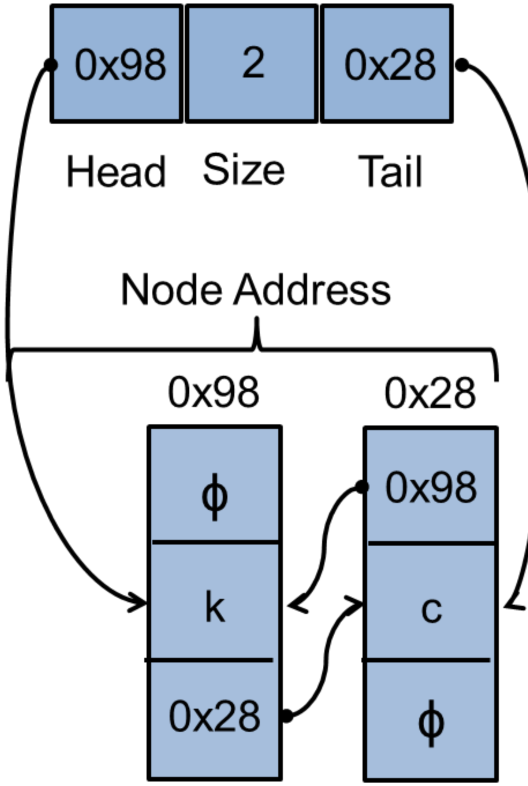
- **Example:**
  - insert('c',0) =>
    insert(‘b',0) =>
    insert(‘a',0) =>
    insert(‘k',1)



Linked List Implementation /3

- `void delete(int position):`
  1. Removes data from the list at the given **`position`**
  2. On deletion, all of the existing data after **`position`** is moved towards the list front by one

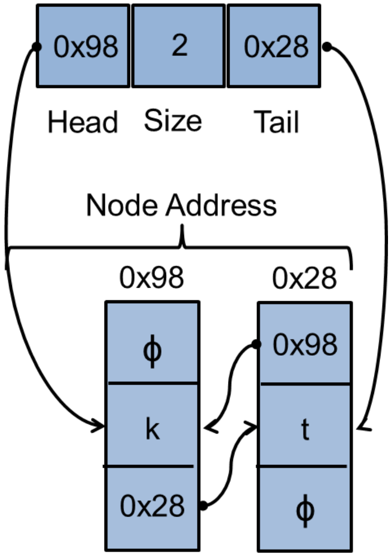
- **Example:**
  - delete(2) => delete(0)



Linked List Implementation /4

- **`DataType select(int position):`**
  - Returns the value stored at the given **`position`**
  - Implemented by iterating through the list until **`position`**

- **`void replace(int position, DataType value):`**
  - Replaces with **`value`** the data stored at **`position`**
  - Implemented by iterating through the list until **`position`** and replacing its value
  - Example: replace(1, ‘t’)



- **`int size():`**
  - Returns the number of elements in the list
  - Implemented by returning **`size`**

Sequential vs. Linked List Implementation

- **Sequential list implementation properties:**
  - Fast access, slow insert/delete
  - Fast insertion at the end, but has a capacity that is limited by the size of its underlying array
    - Later: dynamically growable arrays
  - Processor cache optimization

What exactly is "fast/slow"?? We will talk about this in ~2 weeks ☺

- **Linked list implementation properties:**
  - Slow access, fast insert/delete (if set up efficiently)
  - Its size is more easily adjusted
  - Fragmented memory allocation
  - Increased overall memory requirements (prev/next)

- **Discussion question:**
  - What are other advantages and disadvantages of each implementation choice?

Food for Thought

- **Read:**
  - Chapter 2 (Linked Data Structures) from the course handbook
  - Chapter 3 (Lists) from the course handbook

- **Additional Readings:**
  - Chapter 5 (Linked Lists) from "Data Structures and Other Objects Using C++" by Main and Savitch