The University Of Michigan

EECS402 Lecture 15

Andrew M. Morgan

Savitch Ch. 17
Linked Data Structures

Stacks
Queues
Priority Queues
Double-linked list

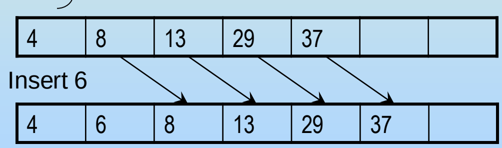
1

Sorted Arrays As Lists

- Arrays are used to store a list of values
- Arrays are contained in contiguous memory
  - Recall – inserting a new element in the middle of an array requires later elements to be "shifted". For large lists of values, this is inefficient.

```
void insertSorted(int value, int &length, int list[])
{
  int i = length - 1;
  while (list[i] > value)
  {
    list[i + 1] = list[i];
    i--;
  }
  list[i + 1] = value;
  length++;
}
```

Shifting array elements that come after the element being inserted



Sorted Arrays, Cot'd

- Due to the need to "shift" elements, sorted arrays are:
  - Inefficient when inserting into the middle or front
  - Inefficient when deleting from the middle or front
- However, sorted arrays are very efficient for searching
  - Can always get to "the middle" element - binary search
- Since inserting and deleting are common operations, we need to find a data structure which allows more efficiency
  - Contiguous memory will not work – will always require a shift
  - "Random" placement requires "random" memory locations
  - Dynamic allocation provides "random" locations, and means that the list can grow as much as necessary
  - The maximum size need not be known – ever
    - This is not true for arrays, even dynamically allocated arrays

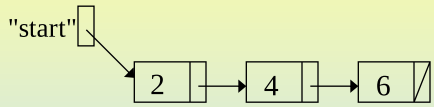
Intro To Linked Lists

- A linked list is a data structure which allows efficient insertion and deletion.
- Consists of "nodes". Each node contains:
  - A value - the data being stored in the list
  - A pointer to another (the next) node
- By carefully keeping pointers accurate, you can start at the first node, and follow pointers through entire list.
- Graphically, linked list nodes are represented as follows:

value	ptr

Linked List Info

- Each node is dynamically allocated, so memory placement is "random"

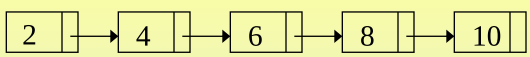


The above linked list <u>may</u> be stored in memory as shown to the right.

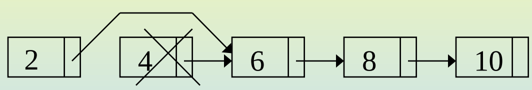
	1000	
	1004	6
	1008	*0
	100C	
	1010	2
	1014	*1030
	1018	
	101C	
"start"	1020	*1010
	1024	
	1028	
	102C	
	1030	4
	1034	*1004
	1038	

Deletion From Linked Lists

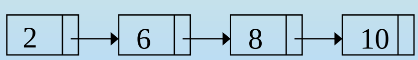
- Given the initial linked list:



Delete node with value 4

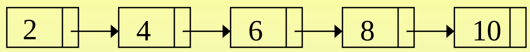


Resulting in

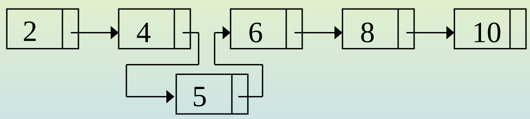


Insertion Into Linked Lists

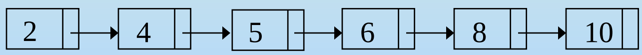
- Given the initial linked list:



Insert node with value 5



Resulting in

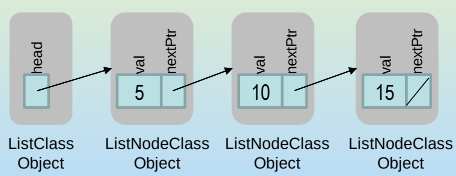


Linked List Implementation Framework

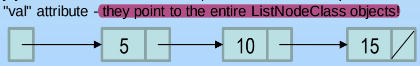

```
class ListNodeClass
{
  private:
    int val;
    ListNodeClass *nextPtr;

  public:
    ListNodeClass(
         const int inVal,
         ListNodeClass* const inNextPtr)
    {
      val = inVal;
      nextPtr = inNextPtr;
    }

    int getVal() const
    {
      return val;
    }

    ListNodeClass* getNextPtr() const
    {
      return nextPtr;
    }
};
```

The position of this?

```
class ListClass
{
  private:
    ListNodeClass *head;
  public:
    ListClass()
    {
      head = 0;
    }
    void insertAtHead(const int valToInsert);
    void printList() const;
    bool deleteFromFront(int &valDeleted);
};
```



Usually just drawn like this, but its important to realize the pointers don't point to the "val" attribute - they point to the entire ListNodeClass objects!



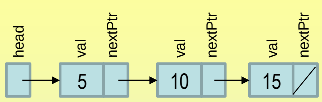
Linked List Functionality

EECS 402

- In the following slides, code is shown that performs several common linked list functions
- This code is meant to go along with a presentation of the algorithms and algorithm development in class

Printing a List (Visiting Each Node)

EECS 402




```
void ListClass::printList() const
{
  ListNodeClass *nodePtr;

  if (head == 0)
  {
    cout << "List is empty!" << endl;
  }
  else
  {
    nodePtr = head;

    cout << "List contents:";
    while (nodePtr != 0)
    {
      cout << " " << nodePtr->getVal();

      nodePtr = nodePtr->getNextPtr();
    }
    cout << endl;
  }
}
```

null

point to the same loc head is pointing to

Can i just use nodePtr -> Val here? NO!!!

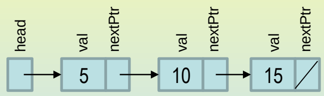
advance the nodeponiter to the next one

Corrected Insert To Front Of List

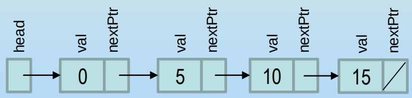
```
void ListClass::insertAtHead(const int valToInsert)
{
  ListNodeClass *nodePtr;

  nodePtr = new ListNodeClass(valToInsert, head);

  head = nodePtr;
}
```

Another advantage of dynamic allocation

"Before":



"Goal":

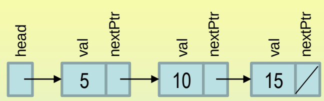


11

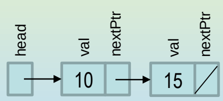
Deleting From Front Of List

```
bool ListClass::deleteFromFront(int &valDeleted)
{
  bool didDeleteItem;
  ListNodeClass *newHeadPtr;

  if (head == 0)
  {
    didDeleteItem = false;
  }
  else
  {
    valDeleted = head->getVal();
    newHeadPtr = head->getNextPtr();
    delete head;
    head = newHeadPtr;

    didDeleteItem = true;
  }

  return didDeleteItem;
}
```

"Before":



"Goal":



12

ListClass Example

EECS 402

- Using the ListClass

```
int main()
{
  ListClass myList;
  int intVal;

  myList.printList();
  myList.insertAtHead(40);
  myList.insertAtHead(30);
  myList.insertAtHead(20);
  myList.insertAtHead(10);
  myList.printList();

  if (myList.deleteFromFront(intVal))
  {
    cout << "Deleted value: " << intVal << endl;
  }
  if (myList.deleteFromFront(intVal))
  {
    cout << "Deleted value: " << intVal << endl;
  }
  myList.printList();

  return 0;
}
```

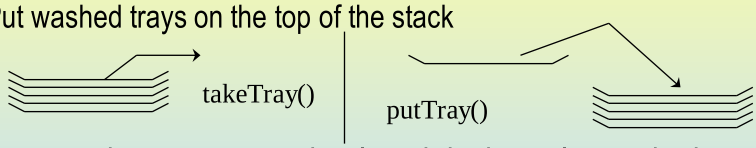
```
List is empty!
List contents: 10 20 30 40
Deleted value: 10
Deleted value: 20
List contents: 30 40
```


The Stack Linked Structure

EECS 402

- A stack is another data structure
  - Used to organize data in a certain way
- Think of a stack as a stack of cafeteria trays
  - Take a tray off the top of the stack
  - Put washed trays on the top of the stack

Last in first out (LIFO)



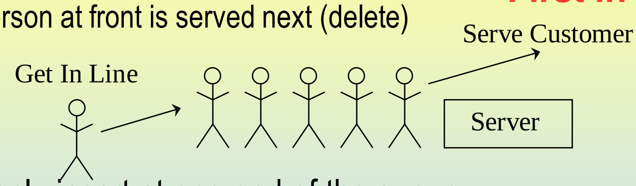
- Bottom tray is not accessed unless it is the only tray in the stack.
- Since only the top of a stack can be accessed, there needs to be only one insert function and one delete function
  - Inserting to a stack is usually called "push"
  - Deleting from a stack is usually called "pop"

The Queue Linked Structure

EECS 402

- A queue is another data structure.
- Think of a queue as a line of people at a store
  - Get into the line at the back (insert)
  - Person at front is served next (delete)

First in first out (FIFO)



- Can only insert at one end of the queue.
  - Inserting to a queue is usually called "enqueue()"
  - "Get In Line" in above diagram
- Can only remove at the other end of the queue
  - Removing from a queue is usually called "dequeue()"
  - "Serve Customer" in above diagram

Queue and Priority Queue are more used

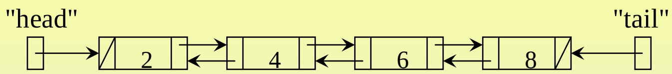
The Priority Queue Linked Structure

EECS 402

- A priority queue works slightly differently than a "normal" queue as described earlier
- Elements in a priority queue are sorted based on a priority
  - Queue order is not dependent on the order in which elements were inserted, as it was for a normal queue
  - As elements are inserted, they are sorted such that the element with the highest priority is at the beginning of the priority queue
  - When an element is removed from the priority queue, the first element (highest priority) is taken, regardless of when it was inserted
  - Elements of the same priority are maintained in the order which they were inserted
- Using a priority queue in which all elements have the same priority is equivalent to using a "normal" queue

The Doubly-Linked List Structure

- The linked list examples we've seen so far have only one pointer
- Often, it may be advantageous to have a node contain multiple pointers




```
class DoublyLinkedListNodeClass
{
  DoublyLinkedListNodeClass *prev;
  int val;
  DoublyLinkedListNodeClass *next;
};

class DoubleLinkedList
{
  private:
    DoublyLinkedListNodeClass *head;
    DoublyLinkedListNodeClass *tail;
  public:
    //...
```

Downside:
1. Use more memory space
2. Code complexity
However, most of the time we would do this cuz it's more convenient