CSC 222: Computer Programming II

Spring 2004

Pointers and linked lists

- human chain analogy
- linked lists: adding/deleting/traversing nodes
- Node class
- linked lists vs. vectors
- stack & queue implementations

List implementations

in List, SortedList, stack, queue, …

- we needed to store a sequence of items
- if size is known ahead of time, can use an array
- better (more flexible) choice is vector
- create with initial size; if need more space, double size
- note: additions not constant time, may result in half the space being wasted

yet another alternative: linked list

- allocate space for each new item as needed
- deallocate space when no longer needed
- want constant time additions/deletions, but not necessarily direct access

In-class exercise

```
class Person {
  public:
    int leftHand;
    Person * rightHand;
}
```

idea: chain people together

- left hand stores data (here, an integer)
- right hand points to the next person in the chain

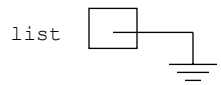
to construct a chain…

- need a pointer (`Person *`) to the front of the chain
- the end of the list must be clear – use a special pointer (NULL) to mark the end

ADD TO FRONT, DELETE FROM FRONT, DISPLAY ALL, SEARCH, …

Linked list code

```
Person * list = NULL;
```



attempting to dereference a NULL pointer results in a run-time error

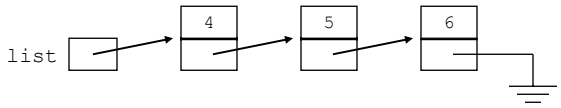
to add to the front of an empty list:

```
list = new Person;              // allocate new Person &
                                // make list point to it
(*list).leftHand = 3;           // store 3 in left hand
(*list).rightHand = NULL;       // store NULL in right hand
```

parenthesization is ugly – better notation:

```
list->leftHand = 3;             // -> is shorthand for
list->rightHand = NULL;         // dereference then access
```

Displaying the contents of a linked list



naïve attempt:

```
while (list != NULL) {
    cout << list->leftHand << endl;
    list = list->rightHand;
}
```

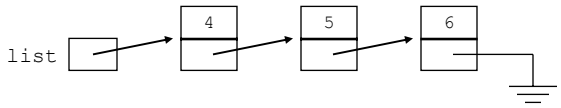
PROBLEMS?

if you lose the only pointer to a node, it becomes inaccessible! better:

```
Person * step = list;
while (step != NULL) {
    cout << step->leftHand << endl;
    list = list->rightHand;
}
```

or could use a for loop

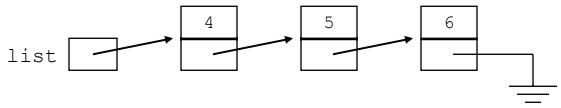
Adding to the front



in general, adding to the front of an arbitrary list:

```
Person * temp = new Person;    // allocate new Person

temp->leftHand = 3;            // store the data

temp->rightHand = list;        // make new Person point
                               // to the old front
list = temp;                   // make new Person the
                               // front
```

Deleting from the front



naïve attempt:

```
list = list->rightHand;
```

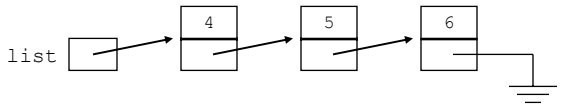
PROBLEMS?

memory leak! better:

```
Person * temp = list;

list = list->rightHand;

delete temp;
```

STILL HAS POTENTIAL PROBLEMS
- what are they?
- fixes?

Adding/deleting in the middle



must find the location (traverse from the front)

- to delete a node from the middle, need a pointer to the node before it.

```
Person * temp = previous->rightHand;
previous->rightHand = temp->rightHand;
delete temp;
```

- to add a node in the middle, need a pointer to the node before it.

```
Person * temp = new Person;
temp->leftHand = ???;
temp->rightHand = previous->rightHand;
previous->rightHand = temp;
```

Node class

```
template <class Item> class Node
{
  public:
    Node<Item>(Item item, Node<Item> * ptr = NULL)
    {
        value = item;
        next = ptr;
    }

    void setValue(Item item)
    {
        value = item;
    }

    void setNext(Node<Item> * ptr)
    {
        next = ptr;
    }

    Item getValue()
    {
        return value;
    }

    Node<Item> * getNext()
    {
        return next;
    }
  private:
    Item value;
    Node<Item> * next;
};
```

can define a generic Node class – useful for constructing any linked list structures

```
Node<int> * list = new Node<int>(12, NULL);

Node<int> * temp = new Node<int>(9, list);
list = temp;

Node<int> * second = list->getNext();
second->setNext(new Node<int>(100, NULL));
```


Linked lists vs. vector

advantages of linked lists

- no empty entries – uses space proportional to the # of items in the list
- once the place is found, can add/delete in constant time
- unlike resizing, all adds/deletes require equal time

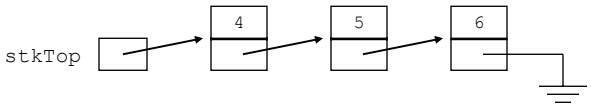
disadvantages of linked lists

- each node wastes space (extra pointer field)
- sequential access only – no direct access, can traverse in 1 direction only

there are applications where linked lists are clear winners

- stack: no need to search for location – add/delete at one end
- queue: no need to search for location – add at one end, delete at other

Implementing stack using a linked list



data fields:

```
Node<Type> * stkTop;
int numItems;
```

```
void push(Type item)
{
   Node<Type> * newNode = new Node<Type>(item, stkTop);
   stkTop = newNode;
   numItems++;
}
```

```
void pop()
{
    if (!empty()) {
        Node<Type> * temp = stkTop;
        stkTop = stkTop->getNext();
        delete temp;

        numItems--;
    }
}
```

```
Type top() const
{
    return stkTop->getValue();
}
```

```
bool empty() const
{
    return (size() == 0);
}
```

```
int size() const
{
    return numItems;
}
```

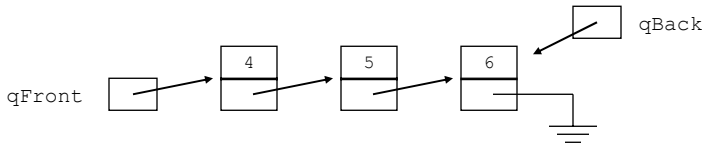

Stack implementation

```
template <class Type> class stack
{
  public:
    stack<Type>() { /* CONSTRUCTOR */ }
    stack<Type>(const stack<Type> & other) { /* COPY CONSTRUCTOR */ }
    ~stack<Type>() { /* DESTRUCTOR */ }
    stack<Type> & operator=(const stack<Type> & other) { /* ASSIGNMENT OPERATOR */ }

    void push(Type item) { /* … */ }
    void pop() { /* … */ }
    Type top() const { /* … */ }
    bool empty() const { /* … */ }
    int size() const { /* … */ }
  private:
    template <class Item> class Node
    {
        /* CLASS DEFINITION */
    };

    Node<Type> * stkTop;
    int numItems;

    Node<Type> * copyList(Node<Type> * ptr) const { /* HELPER */ }
};
```

note: Node class is defined in the private section – not even visible to client program

view full source code

Implementing queue using a linked list



data fields:

```
Node<Type> * qFront;
Node<Type> * qBack;
int numItems;
```

which end is which? does it matter?

```
void push(Type item)
{
  if (empty()) {
    qBack = new Node<Type>(item, NULL);
    qFront = qBack;
  }
  else {
    qBack->setNext(new Node<Type>(item, NULL));
    qBack = qBack->getNext();
  }
  numItems++;
}
```

```
void pop()
{
  if (!empty()) {
    Node<Type> * temp = qFront;
    qFront = qFront->getNext();
    delete temp;

    if (qFront == NULL) {
      qBack == NULL;
    }
    numItems--;
  }
}
```


Queue implementation

```
template <class Type> class queue
{
  public:
    queue<Type>() { /* CONSTRUCTOR */ }
    queue<Type>(const queue<Type> & other) { /* COPY CONSTRUCTOR */ }
    ~queue<Type>() { /* DESTRUCTOR */ }
    queue<Type> & operator=(const queue<Type> & other) { /* ASSIGNMENT OPERATOR */ }

    void push(Type item) { /* … */ }
    void pop() { /* … */ }
    Type front() const { /* … */ }
    bool empty() const { /* … */ }
    int size() const { /* … */ }
  private:
    template <class Item> class Node
    {
        /* CLASS DEFINITION */
    };

    Node<Type> * qFront;
    Node<Type> * qBack;
    int numItems;

    Node<Type> * copyList(Node<Type> * ptr) const { /* HELPER */ }
};
```

again: Node class is defined in the private section – not even visible to client program

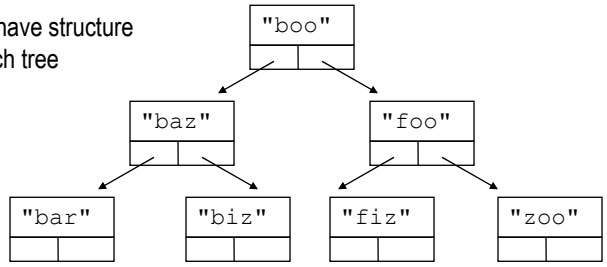
view full source code

Variations on linked lists

can add extra links to make linked lists more flexible

- doubly-linked list has links going both ways – can traverse either direction
- circularly-linked list has link connecting end back to front – can go around
- can combine and have doubly- and circularly-linked list

- non-linear list can have structure
  e.g., binary search tree



in CSC427, we will develop your programming/design skills further

- more complex structures for storing and accessing information
- algorithms for solving more complex problems
- problem-solving approaches

FINAL EXAM

similar format to previous tests

- true or false
- discussion/short answer
- explain/modify/write code

cumulative, but will emphasize material since Test 2

designed to take 60-75 minutes, will allow full 100 minutes

study hints:

- review lecture notes
- review text
- look for supplementary materials where needed (e.g., Web search)
- think big picture -- assimilate the material!
- use online review sheet as a study guide, *but not exhaustive*