

# Linked Data Structures

Wagner Truppel  
Lecturer, Dept. of Computer Science & Engineering  
UC Riverside

[wagner@cs.ucr.edu](mailto:wagner@cs.ucr.edu)  
<http://www.cs.ucr.edu/~wagner>

<http://www.cs.ucr.edu/cs12>

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 1

---

---

---

---

---

---

---

---

# Recall arrays

- Recall arrays
  - ◆ Very convenient
  - ◆ Easy and fast access to elements
  - ◆ But cannot be resized once defined or, if we want to resize them, we need to create a new array and copy all elements from the old to the new array
    - \* This creates additional problems. Ex: could run out of memory in the process of duplicating the old array
- Can we have something that can be resized automatically without much effort?

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 2

---

---

---

---

---

---

---

---

# Linked lists

- Can we have something that can be resized automatically without much effort?
- Yes, they're called **linked lists**.
- They:
  - ◆ Are as convenient as arrays
  - ◆ Provide relatively easy and fast access to elements
  - ◆ Resize themselves automatically without any significant copying
- So... what's the catch?
- The catch is that access to elements is not as fast as it is for arrays

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 3

---

---

---

---

---

---

---

---

# Nodes

```

class Node
{
private:
    int stuff; // for now, let's make it an int
    Node * next; // why a pointer and
                // not the object itself?
public:
    Node(int data, Node * nextNode);
    // other member functions
};
typedef Node * NodePtr; // pointer to a Node
    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 4

---

---

---

---

---

---

---

---

---

---

- The **linked list** is the entire structure.
- The **head** of the list is a **pointer** to the first node, so it's declared as follows:  
NodePtr head;
- If the list is **empty** (no nodes yet), then **head** must be set to null:  
NodePtr head = NULL; // empty list
- So... how do we create a node?

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 5

---

---

---

---

---

---

---

---

---

---

- So... how do we create a node?
- Since we need **pointers** to **Node** objects, we need to create nodes using **new**:  
Node \* aNodePtr = new Node();
- But **Node** doesn't have a no-argument constructor, so the proper way is:  
Node \* aNodePtr = new Node(?, ??);
- What should we have for the first argument? An integer... so, let's say, 7. That's the **stuff** we want to store in the node we're creating. How about the second argument?

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 6

---

---

---

---

---

---

---

---

---

---

- But `Node` doesn't have a no-argument constructor, so the proper way is:  
`Node * aNodePtr = new Node(?, ??);`
- What should we have for the first argument? An integer... so, let's say, 7. That's the **stuff** we want to store in the node we're creating. How about the second argument?
- It must be a pointer to a `Node`. But we don't have another `Node` yet. So we set it to `NULL` at this point:  
`Node * aNodePtr = new Node(7, NULL);`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 7

---

---

---

---

---

---

---

---

- `Node * aNodePtr = new Node(7, NULL);`
- Now we see why we should have pointers instead of the actual objects as member variables... infinite recursion!
- So...
- `Node * aNodePtr = new Node(7, NULL);`
- This is also ok: (recall typedef)
- `NodePtr aNodePtr = new Node(7, NULL);`
- But, so far we've created only a `Node`. We don't have a list yet. This is the proper way to define a list of only one node:  
`NodePtr head = new Node(7, NULL);`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 8

---

---

---

---

---

---

---

---

## List operations

- Constructor:
 

```
Node::Node(int data, Node* nextNode) :
    stuff(data), next(nextNode)
{ }
```
- What other operations would be useful to have?
- Node insertions
  - At the head `headInsert()`
  - In the middle `insert()`
- Node deletions
  - At the head `deleteFirst()`
  - From the middle `delete()`
- Node search `search()`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 9

---

---

---

---

---

---

---

---

**On the board...**

Flesh out the Node class

Implement the operations here

## List operations

- Node insertions
  - ◆ At the head
    - \* headInsert(NodePtr & head, int data)
  - ◆ In the middle
    - \* insertAfter(NodePtr afterMe, int data)
- Node deletions
  - ◆ At the head
    - \* deleteFirst(NodePtr & head)
  - ◆ From the middle
    - \* delete(NodePtr head, NodePtr nodeToGo)
- Node search
  - \* search(NodePtr head, int targetData)

©2003 WL Truppel
CS 12: Intro. Computer Science II • Lecture 13
10

---

---

---

---

---

---

---

---

## Node

stuff

next

NODE

Node

- int stuff

- Node \*next

+ Node(int data, Node \*nextNode)

+ int getStuff()

+ void setStuff(int data)

+ Node\* getNext()

+ void setNext(Node \*nextNode)

```

class Node
{
private:
    int stuff; // for now, we'll use an int
    Node * next; // pointer to the next node
public:
    Node(int data, Node * nextNode);
    int getStuff();
    void setStuff(int data);
    Node * getNext();
    void setNext(Node * nextNode);
};

Node::Node(int data, Node * nextNode) :
    stuff(data), next(nextNode) {}

int Node::getStuff() { return stuff; }
void Node::setStuff(int data) { stuff = data; }

Node * Node::getNext() { return next; }
void Node::setNext(Node * nextNode) { next = nextNode; }
                    
```

©2003 WL Truppel
CS 12: Intro. Computer Science II • Lecture 13
11

---

---

---

---

---

---

---

---

## headInsert

**Rule of thumb for insertions:**

- (1) Create the new node, if necessary
- (2) Fix where the new node has to point to
- (3) Only then, fix who should point to the new node

```

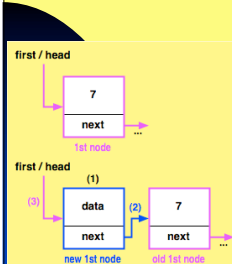
void headInsert(NodePtr & head, int data)
{
    Node * n = new Node(data, NULL); // (1)
    n -> setNext(head); // (2)
    head = n; // (3) without the & above, this
              // line would not change the true head
}
                    
```

More compact, but equivalent, version:

```

void headInsert(NodePtr & head, int data)
{ head = new Node(data, head); }
                    
```

©2003 WL Truppel
CS 12: Intro. Computer Science II • Lecture 13
12




---

---

---

---

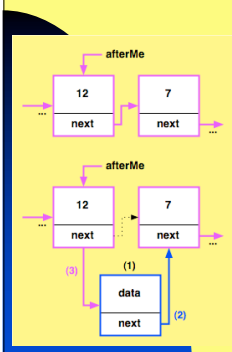
---

---

---

---

## insertAfter



### Rule of thumb for *insertions*:

- (1) Create the new node, if necessary
- (2) Fix where the new node has to point to
- (3) Only then, fix who should point to the new node

```
void insertAfter(NodePtr afterMe, int data)
{
    Node * n = new Node(data, NULL); // (1)
    n -> setNext(afterMe -> getNext()); // (2)
    afterMe -> setNext(n); // (3)
}
```

More compact, but equivalent, version:

```
void insertAfter(NodePtr afterMe, int data)
{
    Node * n = new Node(data,
        afterMe -> getNext());
    afterMe -> setNext(n);
}
```

©2003 WL Truppel

CS 12: Intro. Computer Science II • Lecture 13

13

---

---

---

---

---

---

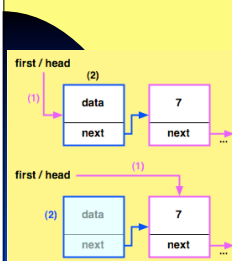
---

---

---

---

## deleteFirst



### Rule of thumb for *head deletions*:

- It's very much the opposite of the rule for insertions!

- (1) Store a pointer to the node to go (1st node) in a local pointer var
- (2) Fix who should point to the node **following** the node to go (ie, fix the head)
- (3) Delete the node to go

```
void deleteFirst(NodePtr & head)
{
    Node * nodeToGo = head; // (1)
    head = ( head -> getNext() ); // (2)
    delete nodeToGo; // (3)
}
```

©2003 WL Truppel

CS 12: Intro. Computer Science II • Lecture 13

14

---

---

---

---

---

---

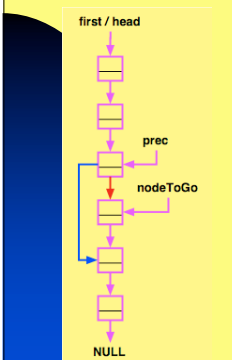
---

---

---

---

## delete



### Rule of thumb for *general deletions*:

- It's very much the opposite of the rule for insertions!
  - But we need to **start from the head** until we find the node **preceding** the node to go.
- (1) Fix who should point to the node **following** the node to go (fix the node **preceding** the node to go)
  - (2) Delete the node to go

```
void delete(NodePtr head, NodePtr nodeToGo)
{
    Node * prec = head;
    while(prec != NULL &&
        (prec -> getNext()) != nodeToGo)
        prec = ( prec -> getNext() );

    if (prec != NULL)
    {
        prec -> setNext( nodeToGo -> getNext() );
        delete nodeToGo;
    }
}
```

©2003 WL Truppel

CS 12: Intro. Computer Science II • Lecture 13

15

---

---

---

---

---

---

---

---

---

---

### search

```

NodePtr search(NodePtr head,
               int targetData)
{
    Node * n = head;
    while(n != NULL &&
          ( n -> getStuff() ) != targetData)
    {
        n = ( n -> getNext() );
    }

    return n;
}
    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 16

---

---

---

---

---

---

---

---

### Node<T>

Node<T>
- T data
- Node<T>* next
+ Node(const T & data, Node<T>* nextNode)
+ const T getData() const
+ void setData(const T & theData)
+ Node<T>* getNext() const
+ void setNext(Node<T>* nextNode)

```

template<class T>
class Node
{
private:
    T data; // any type we care to use !
    Node<T>* next;
    // pointer to the next node
public:
    Node(const T & theData,
         Node<T>* nextNode);
    const T getData() const;
    void setData(const T & theData);
    Node<T>* getNext() const;
    void setNext(Node<T>* nextNode);
}
    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 17

---

---

---

---

---

---

---

---

### Node<T>

```

template<class T>
Node<T>::Node(const T & theData,
              Node<T>* nextNode) :
    data(theData), next(nextNode) {}

template<class T>
const T Node<T>::getData() const
{ return data; }

template<class T>
void Node<T>::setData(const T & theData)
{ data = theData; }

template<class T>
Node<T>* Node<T>::getNext() const
{ return next; }

template<class T>
void Node<T>::setNext(Node<T>* nextNode)
{ next = nextNode; }
    
```

Node<T>
- T data
- Node<T>* next
+ Node(const T & data, Node<T>* nextNode)
+ const T getData() const
+ void setData(const T & theData)
+ Node<T>* getNext() const
+ void setNext(Node<T>* nextNode)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 18

---

---

---

---

---

---

---

---

## List<T>

```

template<class T>
class List
{
    class Node
    { /* ... */ }

private:
    Node<T> * head;
    int size; // number of elements in the list

public:
    List(); // creates an empty list
    bool isEmpty() const;
    int getSize() const;
    Node<T> * getFirst() const;
    Node<T> * getByIndex(int idx) const;
    void addAfter(Node<T> * afterMe, const T & theData);
    void remove(Node<T> * nodeToGo);
    Node<T> * find(const T & theData) const;
}
    
```

**List<T>**

- Node<T> \* head
- int size
- + List()
- + bool isEmpty() const
- + int getSize() const
- + Node<T> \* getFirst() const
- + Node<T> \* getByIndex(int idx) const
- + void addAfter(Node<T> \* afterMe, const T & theData)
- + void remove(Node<T> \* nodeToGo)
- + Node<T> \* find(const T & theData) const

**List::Node<T>**

- T data
- Node<T> \* next
- + Node(const T & data, Node<T> \* nextNode)
- + const T getData() const
- + void setData(const T & theData)
- + Node<T> \* getNext() const
- + void setNext(Node<T> \* nextNode)

©2003 WL Truppel      CS 12: Intro. Computer Science II • Lecture 13      19

---

---

---

---

---

---

---

---

## Doubly-Linked List

- Main advantages:
  - ◆ Now we can have
    - addBefore() in constant time
    - remove() in constant time
    - Bidirectional traversal of the list
- Main disadvantage:
  - ◆ Need to store 2 pointers per node, rather than only 1

©2003 WL Truppel      CS 12: Intro. Computer Science II • Lecture 13      20

---

---

---

---

---

---

---

---

## Data Structures

- Using **Arrays**, **Lists**, and **Trees**, we can build other, specialized, data structures
  - ◆ Stacks
  - ◆ Queues
  - ◆ Heaps
  - ◆ Binary Search Trees
  - ◆ Sets
  - ◆ Maps
  - ◆ Hashtables
  - ◆ And many more !!
- Those you will meet in CS 14 !

©2003 WL Truppel      CS 12: Intro. Computer Science II • Lecture 13      21

---

---

---

---

---

---

---

---

## STL

- C++ has a set of classes implementing all those data structures
- It's called the **Standard Template Library (STL)**
- In CS 14 you'll learn how they are implemented
- Savitch's book covers a couple of the STL classes, such as **vector**
- The STL has other important features
  - ◆ Algorithms (sorting, binary search, etc)
  - ◆ Iterators (more on this in CS 14)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 22

---

---

---

---

---

---

---

---

## Last few lectures

- We'll be reviewing the most important material
  - ◆ Arrays
  - ◆ Pointers
  - ◆ References
  - ◆ Classes
  - ◆ Inheritance
  - ◆ Templates
  - ◆ Lists
- **Bring questions !!!!**

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 23

---

---

---

---

---

---

---

---

## The End... ?

- This is basically **it** for CS 12
- Hope you've enjoyed it
- See you next quarter in CS 14
- Now, please pay my agent before leaving the premises...  
<|8) <|8) <|8) <|8) <|8) <|8) <|8)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 13 24

---

---

---

---

---

---

---

---