Linked Lists

Linked Lists

- The most frequent application of pointers in data structures is to hold at least one pointer area to itself. Thus, it is possible to link one cell to another, much like a train joins a number of cars.
- The **linked list** with analogy to the train is important to the study of data structures.
- What are the advantages of this over array's?

Linked Lists

- A *linked list* is a list of objects held in dynamically allocated memory, and connected to each other by some element such as a pointer.
- A single element of a linked list is called a *node*.
- Using dynamic memory makes linked lists flexible, allowing the program to add new nodes at any time.
- But this also means that two nodes don't necessarily sit right next to each other in memory. For this reason, each node includes a pointer which tells it where to find the next node.

Pointer Functions

- **New:** procedure to allocate memory for a data item and to place its address into *p.*
- **Delete:** procedure to free memory for data item whose address is in p and to make p undefined

- To test the equality of two pointers (p and q), a boolean expression is used : **p = = q**. This tests whether *p* and *q* hold the same address.


```
int * p;
p = NULL;
p = new int;
```

allocates space for one cell of type *int* and places its address in *p*.

p→ **5496** ?

1488 5496

Nodes

```
typedef struct node *pointer;
struct node {
        int       el;
        pointer   next;
        };
pointer      p,q;
```

- Notice the indirect recursion.
- Upon declaration of a pointer variable *p*, there is room in *p* for an address.

- Notice that the node has two elements,
  - an information member
  - a pointer member.
- As with all structures, the individual elements are referenced using a the dot notaton and the element identifier.
- However, there is no structure identifier in this example, therefore the pointer variable *p* along with the operator * is used.

- The use of parentheses is overrides the higher priority of the period operator,as follows :

  `(*p).el`
- Preferably, the structure member operator (→) is used to access the information field :

  `p → el`
- Similarly, the pointer field is accessed as follows

  `p → next`

Example

```
p →el = 5;
p →next = NULL;
```

assigns 5 to the information part (el), pointed to by the pointer variable p, and NULL is the address pointed to by the pointer member(next).

Placing info in a node

```
pointer MakeNode(int e)
{ pointer p = new node;
  if (!p)
     {
     cerr <<''Error making node \n'';
     exit(1);
     } else {
          p → el = e;
          p → next = NULL;
          }
       return p;
     }
```


Delete

`delete p;`

Reverses the effect of *new* by de-allocating memory space. If the link is broken by reassigning *p*, such as:

`p = q;` `or` **`p = NULL;`**`or` **`p = new node_t;`**

the memory cell is still allocated but cannot be reached. To avoid abandoning the cell, the cell should be freed before performing any of the other operations.


```
typedef struct node *pointer;
struct node
  {
  int data;
  pointer next;
  };
```

```
struct node
  {
  int data;
  struct node * next;
  };
```


Specification of ADT List

Structure :

- Finite sequence of nodes. A special position in the list, called the current position, indicates a node, called the current node. The current position is empty if the current position does not indicate a node.
- What functions do we need on this list?

InitList(L) Procedure to initialise the list L to empty

DestroyList(L)

Procedure to make an existing list L empty

ListIsEmpty(L) → b

Boolean function to return TRUE if L is empty

ListIsFull(L) → b

Boolean function to return TRUE if L is full

CurIsEmpty(L) → b

Boolean function to return TRUE if the current position in L is empty

ToFirst(L)

Procedure to make the current node the first node in L; if the list is empty, the current position remains empty.

AtFirst(L) → b

Boolean function to return TRUE if the current node is the first node in the list or if the list and the current position are both empty.

AtEnd(L) → b

Boolean function to return TRUE if the current node is the last node in the list or if the list and the current position are both empty

Advance(L)

Procedure to make the current position indicate the next node in L; if the current node is the last node the current position becomes empty

Insert(L,e)

Procedure to insert a node with information *e* before the current position or, in case L was empty, as the only node in L; the new node becomes the current node;

InsertAfter(L,e)

Procedure to insert a node with information *e* into L after the current node without changing the current position; in case L, make a node containing *e* the only node in L and the current node;

Delete(L)

Procedure to delete the current node in L and to have the current position indicate the next node; if the current node is the last node the current position becomes empty;

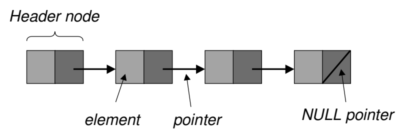
StoreInfo(L,e)

Procedure to update the information portion of the current node to contain e; assume the current position is nonempty

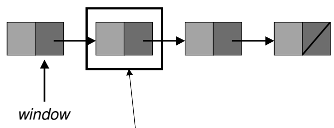
RetrieveInfo(L) → e

Function to return the information in the current node; assume the current position is nonempty

LIST: Linked-List Implementation

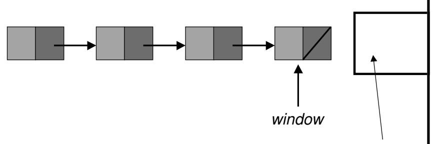


LIST: Linked-List Implementation



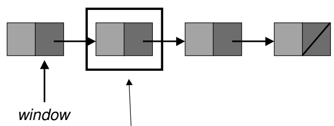
To place the window at this postion we provide a link to the previous node (this is why we need a header node)

LIST: Linked-List Implementation



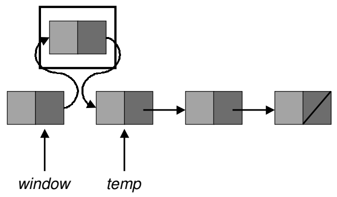
To place the window at end of the list we provide a link to the last node

LIST: Linked-List Implementation



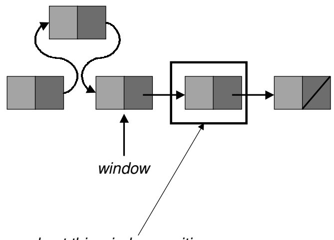
To insert a node at this window position we create the node and re-arrange the links

LIST: Linked-List Implementation



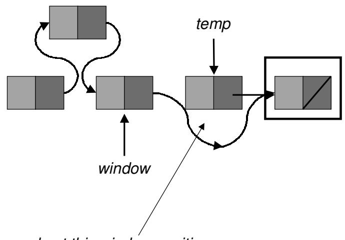
To insert a node at this window position we create the node and re-arrange the links

LIST: Linked-List Implementation



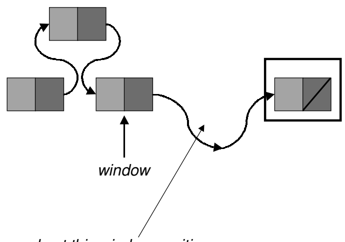
To delete a node at this window position
we re-arrange the links and free the node

LIST: Linked-List Implementation



To delete a node at this window position
we re-arrange the links and free the node

LIST: Linked-List Implementation



To delete a node at this window position
we re-arrange the links and free the node

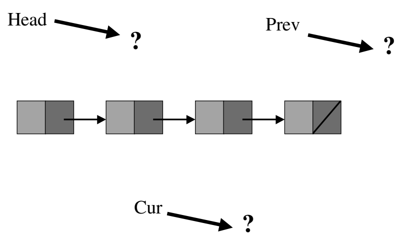
LIST: Linked-List Implementation

```
typedef struct node *pointer;

typedef struct node{
          int el;
          pointer next;
        } NODE;

pointer head;
pointer cur;
pointer prev;
```


LIST: Linked-List Implementation



Initialising a list

```
void InitList()
  {
    head = NULL;
    cur  = NULL;
    prev = NULL;
  };
```

ListIsEmpty?

```
int ListIsEmpty()
{
  if (head = = NULL)
  { return (1);
  }
 else
  { return (0);
  }
};
```

SE 202 Data Structures Rosemary Monahan , NUIM

CurIsEmpty?

```
int CurIsEmpty()
  {
  if (cur = = NULL)
    { return (1);
    }
   else
    { return (0);
    }
  };
```

SE 202 Data Structures Rosemary Monahan , NUIM

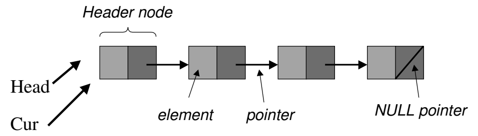
```
void ToFirst()
  {
  cur = head;
  prev = NULL;
  };

int AtFirst()
{
if (head = = cur)
{ return (1);}
else { return (0);}
};
```

SE 202 Data Structures Rosemary Monahan , NUIM

Linked-List Implementation

AtFirst



SE 202 Data Structures Rosemary Monahan , NUIM

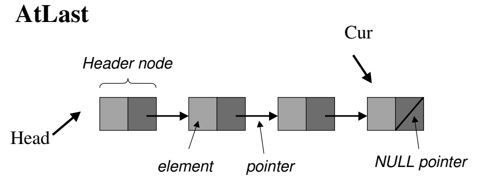
AtLast?

```
int AtLast ()
{ if (CurIsEmpty())
 {return (0); }
   else {if (cur→next == NULL )
           {return (1);}
           else{return (0);}
        }
};
```

SE 202 Data Structures Rosemary Monahan , NUIM

Linked-List Implementation

AtLast



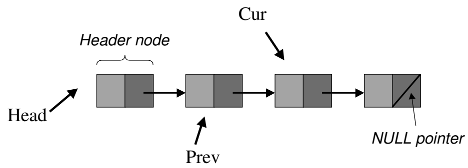
SE 202 Data Structures Rosemary Monahan , NUIM

Advance

```
void Advance()
{
  if (CurIsEmpty())
  cerr << "ERROR : No current node";
  else
     {
     prev = cur;
     cur = cur→next;
     }
};
```


Linked-List Implementation

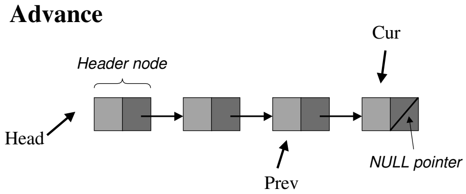
Advance



After Advance this becomes ...

Linked-List Implementation

Advance



Create the node to insert

```
pointer MakeNode(int e)
{ pointer p = new node;
  if (!p)
     {
     cerr <<''Error making node \n'';
     exit(1);
     } else {
          p → el = e;
          p → next = NULL;
          }
       return p;
     }
```


Insert at head

```
void insertathead (int e )
{ pointer p;
  p = MakeNode(e);
  if (head = = NULL)
  { head = p;
    cur = p;
   else    {    p -> next = head;
                head = p;
                cur = p;
           }
  }
```


Inserting ...

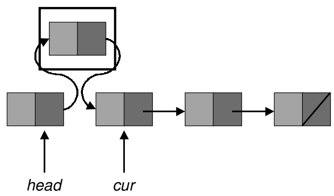
- Before Current:

```
p -> next = cur;
prev -> next = p;
cur = p;
```

- After Current:

```
p-> next = cur -> next;
cur -> next = p;
```

LIST: Linked-List Implementation



To insert a node at this window position we create the node and re-arrange the links

Delete a node

`delete p;`

Reverses the effect of *new* by deallocating memory space.

Warning - don’t lose the link to the rest of the list when you delete a node!

Stacks

- A stack is a special type of list
  - all insertions and deletions take place at one end, called the top
  - thus, the last one added is always the first one available for deletion
  - a dynamic constantly changing object
  - also referred to as
    - pushdown stack
    - pushdown list
    - LIFO list (Last In First Out)

Stack Operations

- *All these operations can be directly implemented using the LIST ADT operations on a List S*
- *Although it may be more efficient to use a dedicated implementation*
- *It depends what you want: code efficiency or software re-use (i.e. utilization efficiency)*

Array Implementation

```
#include<stdio.h>
#define MAX 20

void push(int); /*prototypes*/
int pop();

int stack[MAX], top = 0;
```


Push

```
// add new data to the top of the stack
void push(int I)
{
 if (top >= MAX)
     {
     cout << "The stack is full";
     return;
     }
 else
 {   stack[top] = I;
     top ++;
};
}
```

Pop

```
// remove data from the top of the stack
int pop()
{
 top--;

 if (top < 0)
     {
     cout << "The stack is empty";
     return 0;
     }
 return(stack[top]);
}
```


Underflow & Overflow

- Stack Underflow
  - occurs when we try to pop the empty stack
- Stack Overflow
  - occurs when we try to push something on the full stack
  - this won't happen in a linked list implementation (unless we run out of memory)

Better Representations of stack?

- Could use
  - a structure:
    ```
    struct stack
    { int items[STACKSIZE];
      int top = 0; }
    ```
    with functions pop, push, top,empty …
  - a class with methods pop, push …
  - Data Representation:linked lists, arrays ...

With Structures & Arrays ...

```
void push(stack *ps, int x)
{ if (ps -> top ==STACKSIZE -1)
    { cout << "Stack Overflow";
    }
    else
    { (ps -> top) ++;
     ps -> items[ps -> top] = x;
  }
}
```

Consider using a class and arrays.

What about classes and linked lists?

What do we use stacks for?

- Balanced Expressions
- Converting strings from infix to postfix
- Evaluating postfix expressions
- Storing results in recursive programs
- Removing elements from a list …

Balanced Expressions

- Checking
  - Equal numbers of left and right parentheses
  - Every right parentheses has a matching right
- How?
  - Push opening brackets on the stack
  - Pop matching brackets of the stack
  - If the stack is not empty at the end /we try to pop the empty stack the expression is not balanced

Converting from infix to postfix

- Read in infix string, one item at a time
- If the item is an operand add to the postfix string
- If the item is an operator push it on the stack if

  a) the stack is empty

  b) if the precedence of the item at the top of the stack is < the item being processed
- if the precedence of the item at the top of the stack is > the item being processed pop it to the postfix string
- When the infix string is empty Pop the elements of the stack onto the postfix string to get the result

Evaluating postfix expressions

- Read in string, one item at a time
- If the item is an operand push on the stack
- If the item is an operator pop the stack twice and apply the operand to the resulting values
- Push the result on the stack
- Continue until the string is completed and one value (the answer) remains on the stack.