1806ICT
Programming Fundamentals

Abstract Data Types

Stacks
Linked Lists

1

1

Topics

- ADT Stacks
- Self-Referential Structures
- ADT Linked List
- Basic Operations with Linked Lists
- Variations of linked list
  - Circular linked list
  - Doubly linked list

2

2

The Abstract Data Type (ADT) Stack

- The term ***Abstract Data Type (ADT)*** is used to mean a data structure together with its operations
- The ADT stack is one of the most useful standard data structures
- A stack is a data structure that allows insertion and deletion of data to occur only at a single element, i.e. the top of the stack
  - Last-in-first-out (LIFO) discipline

3

3

The Abstract Data Type (ADT) Stack

- Typical operations with a stack
  - Push: places a value on the stack
  - Pop: retrieves and deletes a value off the stack
  - Top: returns the top value from the stack
  - Empty: tests if the stack is empty
  - Full: tests if the stack is full
  - Reset: clears the stack, or initialises it
- The data structure for stack, along with these operations, is a typical ADT

4

4

The ADT Stack – 1

```
/* An implementation of type stack */
#include <stdio.h>

#define MAXLEN  1000
#define EMPTY   -1
#define FULL    (MAXLEN-1)

typedef struct Stack
{
  char  s[MAXLEN];
  int top;
} stack;

void reset( stack *ptr )
{
  ptr->top = EMPTY;
}

void push( stack *ptr, char c )
{
  (ptr->top)++;
  ptr->s[ptr->top] = c;
}
```

5

5

The ADT Stack – 2

```
char pop( stack *ptr )
{
  char c = ptr->s[ptr->top];
  (ptr->top)--;
  return c;
}

char top( const stack *ptr )
{
  char c = ptr->s[ptr->top];
  return c;
}

int empty( const stack *ptr )
{
  return (ptr->top == EMPTY);
}

int full( const stack *ptr )
{
  return (ptr->top == FULL);
}
```

6

6

The ADT Stack – 3

```
/* Test the stack implementation by reversing a string */
int main()
{
  char str[] = "Programming Fundamentals 1806ICT";
  int    i;
  stack  s;

  reset(&s);              // initialise the stack

  // print original string
  printf("Original string = %s\n", str);

  // push the characters in the string onto the stack
  for (i=0; str[i] != '\0'; i++)
  {
     if (!full(&s))
        push(&s, str[i]);
  }

  // print the string from the stack
  printf("From the stack = ");
  while (!empty(&s))
     printf("%c", pop(&s));

  return 0;
}
```

7

7

Self-Referential Structures

- Is a structure with a member field that points at the same structure type

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

- Each structure is linked to a succeeding structure by using the member **`next`**

8

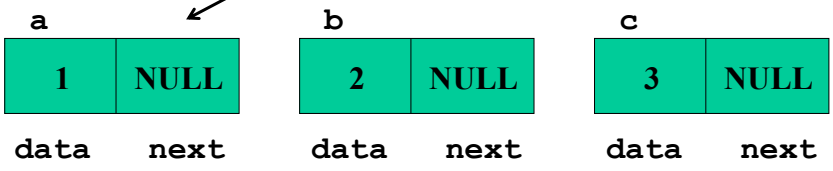
8

Self-Referential Structures

```
struct list
{
    int data;
    struct list *next;
};
struct list a, b, c;

a.data = 1;
a.next = NULL;
b.data = 2;
b.next = NULL;
c.data = 3;
c.next = NULL;
```

3 separate structures



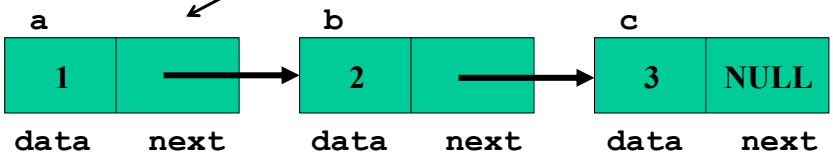
9

9

Self-Referential Structures

```
a.next = &b;
b.next = &c;
```

3 linked structures




```
printf("%d", a.next->data);          2
printf("%d", a.next->next->data);    3
```

10

10

Topics

- ✓ADT Stacks
- ✓Self-Referential Structures
- ADT Linked List
- Basic Operations with Linked Lists
- Variations of linked list
  - Circular linked list
  - Doubly linked list

11

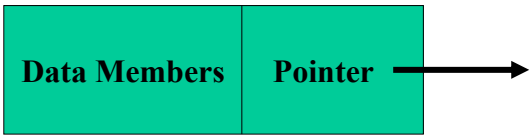
ADT Linked List

- A linked list is a series of *connected nodes* where each node is a *data structure*
- *Dynamically allocated* data structures can be linked together to form a chain
- A linked list can grow or shrink in size as the program runs. This is possible because the nodes in a linked list are dynamically allocated

12

Composition of a Linked List

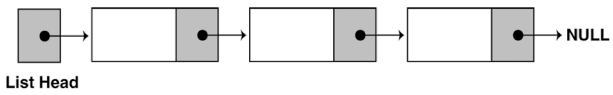
- Each node in the linked list contains
  - One or more members that represent data (e.g. customer names, addresses, telephone numbers, etc.)
  - A pointer, that can point to another node.



13

Composition of a Linked List

- A linked list is called "linked" because each node in the list has a pointer that points to the next node in the list
  - The list head is a pointer to the first node in the list
  - Each node in the list points to the next node in the list
  - The last node points to NULL (the usual way to indicate the end of the list)



14

Creating a Linked List

- Just like any other data type, the information about the node has to first be declared

Step 1) Declare a data structure for the nodes

```
// will use int as data in this example
struct Node
{
      int data;
      struct Node *next;
};
typedef struct Node NODE;
```

15

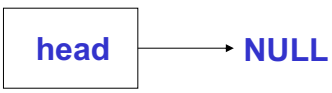
Creating a Linked List

- Next, a pointer needs to be declared to point to the first logical node in the list

Step 2) Declare a pointer to serve as the head of the list:

```
NODE  *head = NULL;
```

Once you have done these 2 steps (i.e. declared a NODE data structure, and created a NULL head pointer), you have an empty linked list



16

Topics

- ✓ADT Stacks
- ✓Self-Referential Structures
- ✓ADT Linked List
- Basic Operations with Linked Lists
- Variations of linked list
  - Circular linked list
  - Doubly linked list

17

17

Linked List Operations

There are 5 basic linked list operations:

- **Appending** a *node* (to the head or to the end)
- **Inserting** a *node* (into a sorted list)
- **Traversing** a *list*
- **Deleting** a *node*
- **Destroying** a *list*

18

Create a New Node

- Allocate memory for the node
- Initialize member contents of the node
- Set pointer to NULL

```
struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node NODE;
```

```
NODE * createNode(int val)
{
        // allocate memory for new node
        NODE *ptr = (NODE *)malloc(sizeof(NODE));
        if (ptr == NULL)
        {
                printf("Fail to create new node\n");
                exit(1);
        }
        else
        {
                ptr->data = val;
                ptr->next = NULL;
        }

        return ptr;
}
```

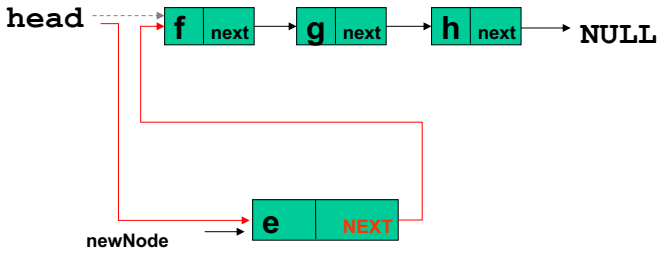
19

19

Appending a Node to the Head

- When appending to the head of the list, the algorithm is

  if the list is empty
  Make the new node the head of the list
  else
  Add the new node to the top of the list
  Make the new node the head of the list
  endif



20

Appending a Node to the Head

```
void insertHead(NODE *newNode)
{
      NODE *ptr = head;

      if (ptr == NULL)  // empty list
      {
            head = newNode;
      }
      else
      {
            newNode->next = head;
            head = newNode;
      }
}
```

21

Appending a Node to the Head

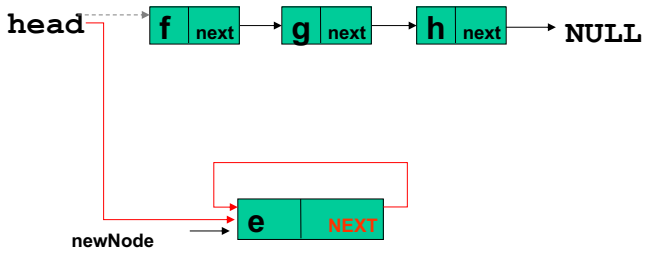
- In the previous algorithm, when appending a node to the head, the order in which the statements are written is extremely important
- What would happen if the statement were written in reverse order, in this manner?

```
head = newNode;
newNode->next = head;
```

22

Appending a Node to the Head

- This would result with the new node pointing to itself, which would make the program go in a never-ending loop when accessing the list
- This would also result in losing access to all the nodes that were currently on the linked list

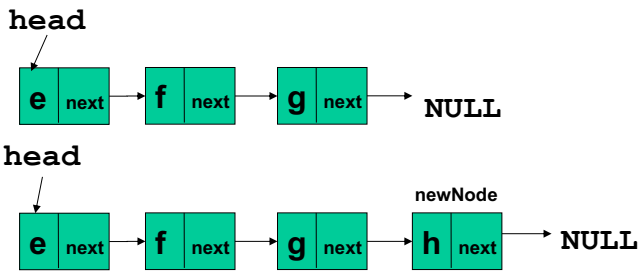


23

Appending a Node to the Tail

- When appending to the end of the list, the algorithm is

```
if the list is empty
    Make the new node the head of the list
else
   Traverse the list to find the last node
   Add the new node to the end of the list
endif
```



24

Appending a Node to the Tail

```
void insertTail(NODE *newNode)
{
    NODE *ptr = head;

    if (ptr == NULL) // empty list
    {
        head = newNode;
    }
    else
    {
        while (ptr->next != NULL)  // traverse list to the end
        {
              ptr = ptr->next;
        }
        ptr->next = newNode;
    }
}
```

25

Traversing a Linked List

- When traversing a list, it is important to remember NOT to move the head pointer from node to node, e.g.

  `head = head->next` 

- This would result in losing access to nodes in the list

- Instead, always assign another pointer to the head of the list, and use that pointer to traverse it

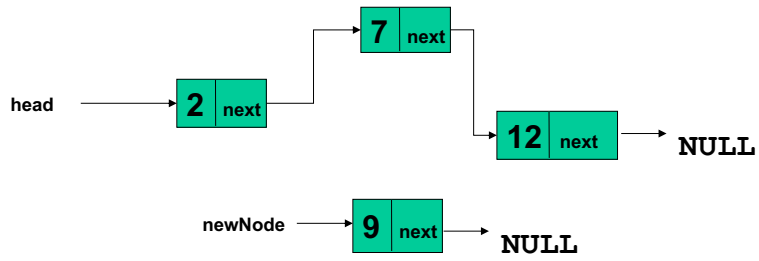
26

Inserting a Node into a Sorted List

if the list is empty
    Make the new node the head of the list
else
    Find the first node whose value is greater than or equal the new value, or the end of the list (whichever is first)
    Insert the new node before the found node, or at end of the list if no node was found
endif

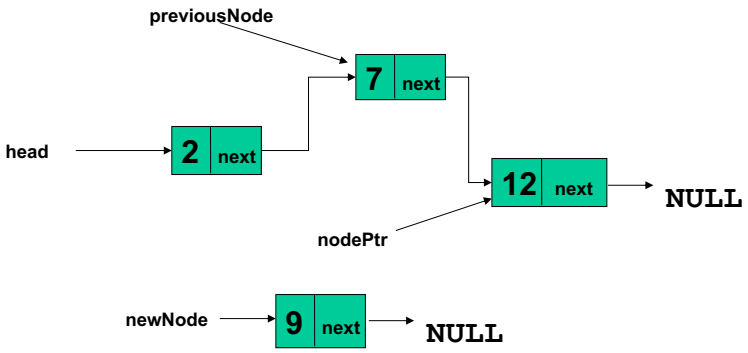
27

Inserting a Node into a Sorted List



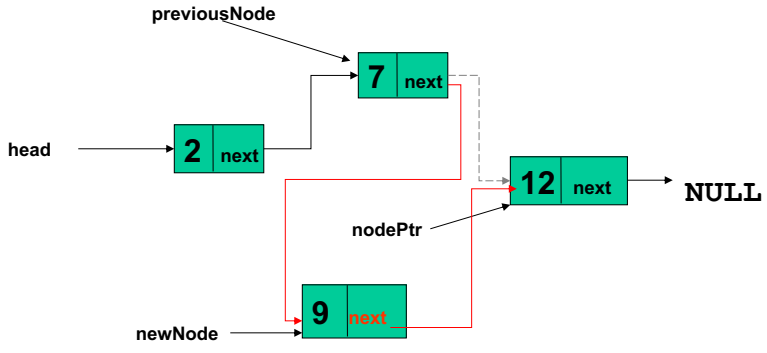
28

Inserting a Node into a Sorted List



29

Inserting a Node into a Sorted List



30

Inserting a Node into a Sorted List

```
void insertNode(NODE *newNode)
{NODE *ptr = head, *prevNode = NULL;
 if (ptr == NULL)                  // new node inserted into empty list
   {head = newNode;
    return;
   }

 // find the first node with higher value than new node
 while ( (ptr != NULL) && (ptr->data < newNode->data))
   {prevNode = ptr;
    ptr = ptr->next;
   }

 if (prevNode == NULL)         // new node has smallest value, insert at head
  {newNode->next = head;
   head = newNode;
  }
 else if (ptr == NULL)         // new node has biggest value, insert at tail
   prevNode->next = newNode;
 else
   {prevNode->next = newNode;
    newNode->next = ptr;
   }
}
```

31

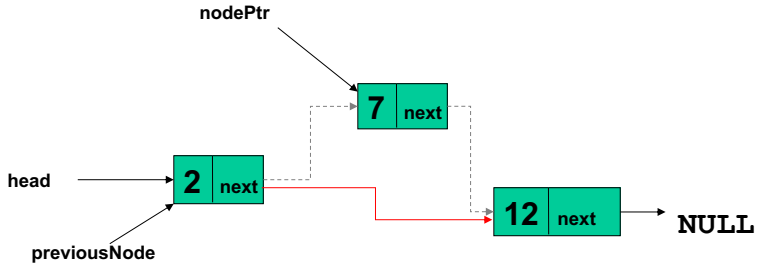
Deleting a Node

- Deleting a node is similar to inserting a node
- The node to be deleted needs to be identified, along with its predecessor. Then
  - Remove the node from the list without breaking the links created by the next pointers
  - Free the node from memory. It is important to remember to free the node so that resources are released

32

Deleting a Node

Assume that we need to delete node that contains the value 7



33

Deleting a Node

```
void deleteNode(int val)
{NODE *ptr = head, *temp = NULL, *prevNode = NULL;

 if (ptr == NULL)    // if empty list, do nothing
   return;
 else if (ptr->data == val)       // head is the node to delete
   {temp = ptr->next;
    free(ptr);
    head = temp;
   }
 else  // search for node to delete
   {while ((ptr != NULL) && (ptr->data != val))
      {prevNode = ptr;
       ptr = ptr->next;
      }
    // link prev node to next node
    if (ptr != NULL)
      {prevNode->next = ptr->next;
       free(ptr);
      }
   }
}
```

34

Traversing the List

- Visit each node in a linked list
  - Print contents, validate data, etc.
- Basic process
  - set a pointer to the head pointer
  - while pointer is not NULL
    process data
    go to the next node by setting the pointer to the pointer field of the current node in the list
    end while

35

35

Traversing the List

```
void printList()
{
      NODE *ptr = head;

      while (ptr != NULL)
      {
            printf("%d ", ptr->data);
            ptr = ptr->next;
      }
      printf("\n");
}
```

36

36

Finding a Node on the List

- Searching for a node on the list with a specific value

```
NODE * findNode(int val)
{
   NODE *ptr = head;

   while (ptr != NULL)
   {
         if (ptr->data == val)
               return ptr;
         else
               ptr = ptr->next;
   }

   return ptr;
}
```

37

37

Destroying the List

- Step through the list and delete each node one by one
  - Free the node from memory
  - Remember to set head to NULL

```
void destroyList()
{
       NODE *ptr = head;
       NODE *nextNode = NULL;

       while (ptr != NULL)
       {
             nextNode = ptr->next;
             free(ptr);
             ptr = nextNode;
       }

       head = NULL;
}
```

38

38

Advantages of Linked Lists over Arrays

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages
  - A linked list can easily grow and shrink in size. The programmer doesn't need to know how many nodes will be in the list. They are created in memory as needed
  - In contrast, the size of an array is fixed at compilation time
  - Speed of insertion or deletion from the list. When a node is inserted, or deleted from a linked list, none of the other nodes have to be moved, only need to reset some pointers
  - In contrast, inserting and deleting elements into and out of arrays requires moving elements of the array, either pushing the entire array down one spot, or shifting all the elements in the array up one spot

39

Topics

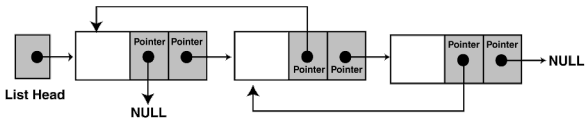
- ✓ADT Stacks
- ✓Self-Referential Structures
- ✓ADT Linked List
- ✓Basic Operations with Linked Lists
- Variations of linked list
  - Circular linked list
  - Doubly linked list

40

40

Variations of Linked Lists

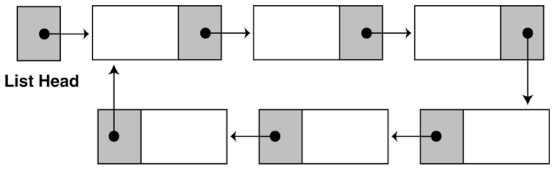
- Doubly linked lists
  - Each node points to not only successor but the predecessor
  - There are two `NULL:` at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



41

Variations of Linked Lists

- Circular linked lists
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

42

Topics

- ADT Stacks
- Self-Referential Structures
- ADT Linked List
- Basic Operations with Linked Lists
- Variations of linked list
  - Circular linked list
  - Doubly linked list