Structures & Pointers

22 February 2017 21:05

Struct	Type of variables	All fields are public		Can initialise all data as you declare a structure
Class	Variables and member functions	Public member - visible from outside	Private by default	

Pointers

- Variable whose value represents a memory location
- Declare in C++ , **varType *NameofPointer**
- Once declared, ***NameofPointer** gives access to variable pointer points to
- To change pointer's actual value, just use **NameofPointer = ..**

- **Always initialise pointers**

- Address of variable in C++ , **address = &NameofVar**

- Arrays
  - arrayName = pointer to start of array
  - &arrayName[N] = memory location of Nth elements in array

- Call-by-reference in C++ done by passing pointer to variable passed

Arguments to main()

- int main (int argc, char *argv[])
- Passed through command line
- Special built-in
  - argc - number of arguments entered
    - Counts from 0, so default is 1?
  - argv - an array of pointers to the arguments
    - argv[0] = &argc

- Syntax
  - ./programName charAgument
    - charArg as it was declared in main above - char * argv[]

Linked Lists

22 February 2017 21:06

Linked lists are a dynamic data structure
Can grow and shrink during program execution

Head of list - pointer to 1st Node
End of list - points to NULL

Advantages
- **Uses only memory needed**
- Computationally **efficient to add/delete data**

Disadvantages
- Memory **overhead** - have to store value of pointers
- **Slow access** to data - have to traverse linear list

addToList function - adds to head of list
- Pass list **head pointer by reference**
- **Create new Node**
    - Assign data
    - Node-**>next** points to **where head was**
- Head now points to new Node

In C++,
- Use a structure
    - Data field - use **typedef** above
    - Another field which is a pointer of type structure (struct in which it is a field of)
- Typedef *structure as StructurePtr - easier for human reading

- Declare a pointer to head of list
    - Create new nodes dynamically using **'new'** syntax

The heap
- Where dynamic data structures are stored
- Use **'delete'** syntax to free up space in heap

- Pointer to link list not in heap,
    - But all Nodes are
    - NULL points to somewhere chosen in the heap

To traverse list

While loop (**pointer != NULL**)
Do action,
Pointer = Node->next

Ordered Lists

22 February 2017 21:06

A link list ordered in a predefined way

<u>Routines</u> **(maintains order)**

- **Insert**
- **Delete**
- **Lookup -** extract info

<u>Insert routine PDL</u>

1. **Create new node**
    a. Assign data
    b. ->next points to NULL

2. **If list is empty**
    a. Set head of list to new node
    Exit

3. If new **element is smaller than 1st Node**
    a. Add element at beginning of list
        i. ->next points to head of list
        ii. Head of list points to new Node

4. Otherwise
    a. Find where to insert item
        i. Use **search and last pointers**
        ii. Use a bool found (= false at start)
        For exiting loop
    b. Insert item
        i. **New->next points to search**
        As search data > data
        ii. Last->next points to new

<u>Delete routine PDL</u>

1. Declare search, last and old pointers
    a. Old used for deleting
    b. Also have bool found
2. **Empty list**
    Exit

3. **Delete from head**
    a. old pointer used
    b. Old = head
    c. Head points to next
    d. Delete old

4. Otherwise
    a. Search for item (while loop - traverse list)
        i. Search and last pointers used
    b. If found (search-> data == data)
        i. Last-> next points to search ->next
        ii. Delete search

Recursion

22 February 2017 21:06

Recursively defined data structures **match well** with recursive routines
Recursive functions make it easier to implement some mathematical functions

Recursive functions have a **base case**

Once base case is reached, functions start to return values in order of Last In First Out - LIFO

Trick :

- Each call gets closer to the base case
- Can always be used instead of a loop
    - Marginally less efficient, as overhead with maintaining the stack

The Stack

Each function is given a **block in the stack** when it is called

- The block of memory contains
    - **Local variables**
    - **Local copies** of parameters passed by value
    - **Pointers** to parameters **passed by reference**
    - **Return address**
        - Where to **start executing again after function terminates**
- This block is emptied after function terminates

Stack shrinks in opposite order it grew with a recursive function

- **LIFO structure**

Hash Tables

22 February 2017 21:06

Advantages

Constant search time - only compute key
- Want to find data, know its key
- Enter key in hash function
- Get index

Hash table:

1. **An Array**
2. **A key**

Idea: map key to a memory location

Key is assigned to a value

Key inserted to hash function

Return index in array

Value stored in this index

Directly addressed - Hash function never generates same index for different keys

Issue - large array

Solution, allows collisions - not directly addressed

Focus on **chained hash table**s in this course

Array index is head of a link list

Implement in C++
- **Declare an array of pointers**

List Processing

22 February 2017 23:38

<u>Amend two lists</u> - 3 methods:

1. Loop
    a. Temp list for list1
    b. Reverse order add to list2
2. Recursion
    a. Base case - list1 has only one element
    b. Keep calling append() with smaller list
3. Update the pointers

1. <u>Loop</u>
 - First create a temp list
    - reverse list of list1
 - Then is adds each element of temp to start if list 2

2. <u>Recursion</u>
    a. Recursive with smaller list1 and list2
        i. Until list1 has no entries -> NULL
    b. Then adds last element of list1 to head of list2
    c. Until stack empty

3. <u>Pointers</u>
    a. Special case when list is empty
        i. Return list2
    b. Otherwise
        i. Find end of first list
        ii. Attach second list

<u>Reversing a list</u>

1. Loop method
2. Recursion method

1. <u>Loop</u> (list != NULL)
    a. Temp -> last node
    b. Delink last node from list

2. <u>Recursion</u>
    a. Uses an accumulating parameter
        i. Gradually build up result in function parameter
        ii. Then return result when base case reached
        iii. On exit calls, this value floats up untouched
    b. Done by
        i. Using recursion
            1) Perform action until base case reached
            2) Base case == reach end
            3) Stop actions at base case
        ii. In function, return function
            1) Progressed list and,
            2) accumulating parament

Comparison

1. Loop
    a. Creates new copy of list1 - followed by copy of list2
    b. tempList is a duplicate of list1 - so tempList should be destroyed
2. Recursion
    a. Links element one by one via stack (passing value)
3. Pointer
    a. Doesn't create any new data
    b. But doesn’t save lists - careful

Binary Trees

14 March 2017 19:57

When ordered, anything to the right is larger, and anything to the left is smaller.

Tree is a dynamic structure, and exists in heap like a link list

Advantages;

- Easy to insert new element
- Easy to traverse tree in order
- Much quicker lookup than link list, O(logn) vs O(n)

To declare, need data field.
AND, two pointer fields for left and right sub-trees.

Insertion - preserve ordering

- Base case - empty sub-tree
  - New Node
    - Sub-trees => NULL
  - Pointer passed now -> new Node
- Else, if less, recursion to the left
- Else, recursion to the right

To Traverse , Use function Recursion with left and right sub-trees,
IF sub-tree != NULL
USING one line to carry out the function, before going to sub-trees

Or, use correct logic before visiting sub trees.

Deletion - preserve Ordering

- Find Node,
- Replace Node data with Node X
  - X is leftmost node in right sub-tree
- Delete Node X

To Print;

- Check NULL
  - Print function left sub-tree
  - Cout
  - Print Function right sub-tree

Code:

1. Check if empty
   a. If found, call delete root function
      i. If right is empty, left is root
      ii. If not, call leftmost
         1) If left of this is NULL, great
            a) And call delete root on this Node
         2) If not, go left again
2. Else, Traverse accordingly, by checking with <

All Node Tree	Leaf Only Tree
Insertion Easy	Insertion Hard
Deletion Hard	Deletion Easy

Balanced Trees

14 March 2017 21:00

Completely Balanced: every node in every layer above the bottom, **has two children**

Balanced: L and R sub tree differ by MORE than 1, BF = -1, 0, +1

Height - Tree - longest path from root to leaf

Depth - Node - distance from specific node to root

At worst, a binary tree requires N operations, average logN

Improve by ROTATION

- **Unordered** tree - **just rebuild**
- **Ordered** - harder - rebuild using **sorting**

BF: height of right - left: HORL

TREE ROTATION

- Local operation
- Changes structure without changing order
- Do when a Node is UNBALANCED
  - Start from lowest node that is unbalanced

Right Rotation

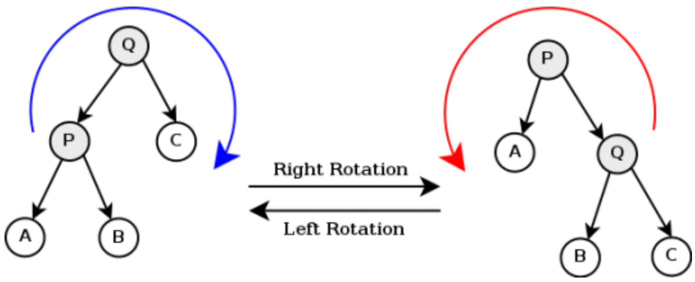
- **Root becomes new right** of the pivot node
- **Previous right** now **left of root's new position**

- DECREASE LEFT side height
- INCREASE RIGHT side height

Left Rotation

- **Root becomes new left** of the pivot node
- **Previous left** now **right of root's new position**

- DECREASE RIGHT side height
- INCREASE LEFT side height



AVL self-balancing tree

- Name from: Adelson-Velsky-Landis
- **Heights of an two sub-tree differ only by one**

- **logN** operations for average and worst case
- Cost - rotations

AVL Insertion

- Update BF
- Check for balance
-

Case	Rotation
LL	**Right**
LR	**Left Right**
RL	**Right Left**
RR	**Left**

Parsing and Expression Evaluation

19 March 2017 21:51

Backus-Naur Form:
- '::=' = is defined
- <> around words
- '|' = OR

Done for computer to carry out operations in correct order
- Produce correct machine instructions

BNF Arithmetic Expressions - Implement BIDMAS - **left-recursive**
- <expression> ::= <term> | <expression> + <term> | <expression> - <term>
- <term> ::= <factor> | <term> * <factor> | <term> / <factor>
- <factor> ::= <number> | <expression>

Lexeme: smallest syntactic unit of a language

Parse Trees

19 March 2017 21:52

Binary tree with lexemes as nodes - excluding brackets
Brackets represented by links BETWEEN nodes of OPERATORS

All **leaves** of the trees contain **Operands**

Struct in C++

- Same as binary with extra fields;
- bool isLeaf
- int number // only if(isLeaf)
- char op // only if(!isLeaf)

Sorting

19 March 2017 21:52

Analyse a Sort by:
- Scalability
- Avg and worse case
- Resources requires

Big-O notation
- As tends to infinity, O(n) is an upper bound
    - **Ignores all constants (including constant multipliers)**
- No description for performance of small n

Bubble Sort:
1. Ends when no swaps performed
2. **STABLE**
3. **Use** for;
    a. **SMALL** lists
    b. Most of list is **ALREADY SORTED**

Merge Sort:
1. Compare pairs of elements, then merge
2. **STABLE**

Heap Sort:
1. Uses a Heap Tree
    a. Parent key > child key
    b. Tree is complete
2. Algorithm:
    a. Build Heap
    b. Remove root and put it at end of list
    c. **Restructure Heap and repeat B**
3. **UNSTABLE**

Quicksort:
1. Divide and Conquer
2. Algorithm:
    a. Pick a pivot

Value of element	Position to P
LESS	LEFT
GREATER	RIGHT

    b. Pick two new pivots either side
3. Stable - DEPENDING UPON SELECTION OF PIVOT

Name	Best	Average	Worst	Extra Memory
Bubble sort	n	n^2	n^2	I Just 1 more memory location
Merge sort	nlogn	nlogn	nlogn	Depends (n)
Heap sort	nlogn	nlogn	nlogn	I
Quicksort	nlogn	nlogn	n^2	logn

Memory additional to storing the list