Introduction to Data Structures

Overview

- Abstract data types and data structures
- Linear data structures
- Arrays
- Stacks
- Queues

Data Type

- Realization of some abstract notion.
- To the system: The way in which a particular memory chunk is interpreted.
- To the user: Data type is identified by its behaviour; the actual storage of data does not alter the behaviour.
- Example
  - Data type int in C++ realizes abstract notion of integers and provides operators like +, -, *, / .

Data Structures

- Conceptually similar to a data type.
- Typically, more general purpose and used to organise specific data.
- Defines set of operators and representation.
- Combines more than one simple data items or data structures.
- Relative positions of the data items are of interest.
- Example:
  - Array, matrix, list…

Data Structures

- User of a data structure(DS) is interested only in the operators provided.
- Implementation details hidden from the application program.
- Higher level of abstraction for the programmer.
- Can be used like a data type in the application.

‘Structure’ of a Typical DS

- DS means relative positioning of data items and their access mechanism.
- The positioning can be implicit (as in the case of arrays) or explicit, i.e. each item holds references to its neighbours.
- This gives two parts per node of a DS
  - Data (can be any data type, independent of the DS)
  - Linkages to neighbouring nodes (specific to the DS)

‘Structure’ of a Typical DS

- The access mechanism defines the type of DS
  - Totally ordered (linear)
  - Partially ordered (eg: trees)
  - Non-linear (eg: graphs)
- Actual representation can be different as long as access mechanism is same.

Linear DS

- Data items (or nodes) arranged in linear fashion.
- Every node has a unique next element and a unique previous element.
- Example
  - Array, linked list, stack, queue, vector.

Array

- Ordered collection of objects of same type.
- Contiguous storage allocation: enables random access.
- Linkages to neighbours are implicit - through position in the memory.
- Size is fixed while constructing.
- Insertion of item at specific position may require shifting of existing items. *(why?)*

Stack

- An opaque pipe closed at one end.
- Only one element at a time is available, no other information is accessible.
- Addition and deletion of elements only at the open end.
- Last in first out (LIFO) behaviour.

Stack Example

Procedure invocation and return

```
A( )          B( )          C ( )
{             {             {
   B( ) ;        C( );         D( ) ;
    :             :             :
    :             :             :
 }            }             }
```

Stack Operations

- clear() – clear/empty the stack
- is_empty() – check if it is empty
- is_full() – check if it is full?
- push(el) – Put the element *el* on the top of the stack. Note: you can't put it anywhere else!
- pop() - remove the topmost element from the stack. Note: can't remove any other element!
- top_el() - Return the topmost element without removing it.

Stack Operations

- push(item) - item becomes the new top element

  push (3) (3)

  push (10) (10, 3)

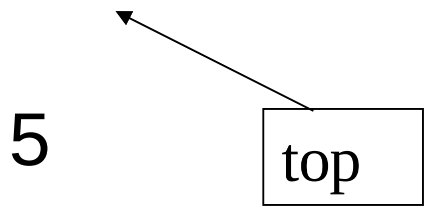
- pop() - top element removed. An error if stack is empty.

  i = pop () (3)

  Value of *i* becomes 10

Stack Implementation

- Stack using array
  - Data items stored as an array
  - Position of ‘top’ maintained by an index



1

Stack Implementation

```
class ArrayStack {
  private:
     int top;   //will point to the topmost element of  the stack
     int size;
     int *storage;
  public:
     ArrayStack(int n){
        size = n;
        storage = new int[size];
        top = -1;
  }
```

Stack Implementation

```
void clear(){top = -1;}
bool is_full(){
        return (top >= size-1);
   }
bool is_empty(){
        return (top == -1);
   }
void push(int el){
        if (!is_full())
            {top++; storage[top] = el;}
     else cout<<"Overflow";
   }
```

Stack Implementation

```
int pop(){
     if(!is_empty()){
          int tmp = storage[top]; top--;
          return tmp;       }
     else
          cout<<"Underflow";
  }
int top_el(){
     if(!is_empty()){return storage[top];}
     else return NULL;
  }
};
```

Stack Application

- Many cases of LIFO(last-in first-out) applications in computer science, particularly programming language implementation, algorithms and operating systems.
- We will discuss the bracket matching problem as a case study.

Bracket Matching Problem

- [([ ))] - invalid
- [ ] ( ) ([ ]) - valid
- [{ } ([ ])] - valid

Conditions:

- Every open bracket should be closed by the appropriate close bracket.
- No extra close brackets.
- The last bracket opened should be closed first.

Pseudo Code

```
valid = true // assume string is valid so far
Stack s = new Stack(); s.clear();
while (valid  & (input not over )){
    read next symbol (symb) ;
    if (symb is '(' or '[' or '{' ){
        s.push(symb);
    }
    else if (symb is ')' or ']' or '}' ){
        if (s.is_empty()) valid = false // too many closing
    brackets
        i = s.pop()
        if (i does not match with symb) valid = false
    }
}
```

Pseudo Code

```
if (! s.is_empty ( ) )
    valid = false ;  //  too many open brackets
if (valid)  print("String is valid");
else print("String is not valid");
```

Queue

- Modeled on real queue at counters, service centers, etc.
- When input is to be processed as per arrival, but can't wait for all input to be ready before processing.
- Insertion at the tail; deletion from the front.
- Unlike real queues, for each deletion we don't want to move all the elements.

Queue Operations

- clear() – Clears the queue
- is_empty()
- is_full()
- enqueue(el) – Puts element *el* at the end of the queue
- dequeue() – Removes the first element from the queue
- first_el() – Returns the first element in the queue without removing it

Implementation Using Arrays

first a b c d last

- ‘*last*’ moves right for every addition
- ‘*first*’ moves right for every deletion
- empty when *first* = *last* + 1
- full when *last* = arraysize
- but ...

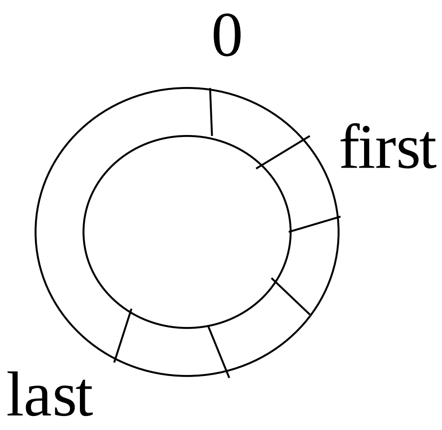
Implementation Using Arrays

- Queue moves right
- Locations freed by delete not reused
- Queue may be flagged full even with a single element!

Implementation Using Circular Arrays

- View array as circular -
  q[max +1] -> q[0];   q[0 - 1] -> q[max]

- *last* and *first* travels clockwise
- *first* need not be less than *last*



Implementation Using Circular Arrays

```
class ArrayQueue{
  private:
int first, last, size, count;
int *storage;
  public:
     ArrayQueue(int n){
     first = 0;last = -1;count = 0;size = n;
     storage = new int[size];
    }
   bool is_full(){
     return (count >= size);
    }
    bool is_empty(){
     return (count == 0);
   }
```

Implementation Using Circular Arrays

```
void enqueue(int el){
        last = (last+1)%size; count++;
        storage[last] = el;
        //add suitable error checks
    }
int dequeue(){
        int tmp = storage[first];
        first = (first+1)%size; count--;
        return tmp;
        //add suitable error checks
  }
};
```

Example

Railway Ticket Reservation

- Ten counters, all identical.
- Customers arrive - Assume only one type of transaction.
- A single queue.
- When a counter falls vacant, the person at the front of the queue goes there.
- Simulate this.

Ticket Counter

- A queue structure will maintain the queue.
- An array serve[1..c] records status of the counters. -1 indicates empty; otherwise expected time of completion.
- For each time tick:
  - if any arrival, insert in the queue.
  - if any counter has serve [i] = timer, set serve[i] = -1
  - if any counter vacant, dequeue a person and send to first available counter, update serve array.
- Can extend to record max queue size, vacant counter time etc.

Summary

- Data Structures: useful abstractions for a programmer
- Arrays, Stacks and Queues: simple linear data structures
- We will discuss “better” representations for these, and introduce more sophisticated DS in subsequent sessions.