Arrays

- Built-in in most programming languages
- Two kinds (programmer responsibility):
  - Unordered: attendance tracking
  - Ordered: high scorers
- Operations:
  - Insertion
  - Deletion
  - Search

Operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unordered</th>
<th>Ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Deletion</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Search</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

Pluses & minuses

- Fast element access; simple
- Impossible to resize; slow deletion
- Many applications require resizing!
- Required size not always immediately available.
Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

Abstract Data Types - Example

An ADT modeling a simple stock trading system
- Data stored: buy/sell orders
- Operations on the data:
  - order buy(stock, shares, price)
  - order sell(stock, shares, price)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Just one method:
  - Object * getElement(): returns the element stored at the position

List ADT

- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()
Singly Linked Lists

✓ A singly linked list is a concrete data structure consisting of a sequence of nodes
✓ Each node stores
  • element
  • link to the next node

Node struct

```c
struct Node {
    // Instance variables
    Object* element;
    Node* next;

    // Initialize a node
    Node() {
        this(null, null);
    }
    Node(Object* e, Node* n) {
        element = e;
        next = n;
    }
```

Singly linked list

```
struct SLinkedList {
    Node* head; // head node of the list
    long size;  // number of nodes in the list

    /* Default constructor that creates an empty list */
    SLinkedList() {
        head = null;
        size = 0;
    }
    // ... update and search methods would go here ...
    boolean isEmpty() {return head ==null; }
    void insertAfter(Node * node, Object* element) {...}
    ...
};
```

Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Make new node point to old head
4. Update head to point to new node
Algorithm addFirst

Algorithm \( addFirst(v) \)

Input node \( v \) to be added to the beginning of the list

Output

\[
\begin{align*}
    v&.setNext\ (head) & \{ \text{make } v \text{ point to the old head node} \} \\
    head &\leftarrow v & \{ \text{make variable } head \text{ point to new node} \} \\
    size &\leftarrow size + 1 & \{ \text{increment the node count} \}
\end{align*}
\]

Removing at the Head

1. Update head to point to next node in the list
2. Disallocate the former first node
   - the garbage collector to reclaim (Java), or
   - the programmer does the job (C/C++)

Singly linked list with ‘tail’ sentinel

```c
struct SLinkedListWithTail {
    Node* head; // head node
    Node* tail; // tail node of the list
}

// Default constructor that creates an empty list */
SLinkedListWithTail() {
    head = null;
    tail = null;
}
```

// ... update and search methods would go here ...

### Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

### Algorithm addLast

Algorithm \textit{addLast}(v)

\textbf{Input} node \(v\) to be added to the end of the list

\textbf{Output}

\(v\).\textit{next} \rightarrow \text{(NULL)}  \quad \{\text{make new node } v \text{ point to null object}\}

\(\text{tail}.\text{next} \leftarrow v\)  \quad \{\text{make old tail node point to new node}\}

\(\text{tail} \leftarrow \text{size}\);  \quad \{\text{make variable tail node point to new node}\}

\(\text{size} \leftarrow \text{size} + 1\)  \quad \{\text{increment the node count}\}

### Removing at the Tail

- Removing at the tail of a singly linked list cannot be efficient!
- There is no constant-time way to update the tail to point to the previous node

### Doubly Linked List

- A doubly linked list is often more convenient!
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
Node struct

/* Node of a doubly linked list of strings */
struct DNode {
    string* element;  // Pointer to the data
    DNode* next, *prev;  // Pointers to next and previous node
}

/* Initialize a node. */
DNode(string* e, DNode* p, DNode* n) {
    element = e;
    prev = p;
    next = n;
}

string* getElement() { return element; }

Class for doubly linked list

struct DList {
    DNode* header, *trailer;  // Sentinels
    long size;  // Number of nodes in the list
}

/* Default constructor that creates an empty list */
DList() {
    header = new DNode(NULL, NULL, NULL);
    trailer = new DNode(NULL, header, NULL);
    header->next = trailer;
    size = 0;
}

// ... update and search methods would go here ...

Insertion

We visualize operation insertAfter(p, X), which returns position q.

Insertion Algorithm

Algorithm insertAfter(p, e):
Create a new node v
v.setElement(e)
{ link v to its predecessor }
v.setPrev(p)
{ link v to its successor }
(p.getNext()).setPrev(v)
{ link p’s old successor to v }
p.setNext(v)
{ link p to its new successor, v }
return v
{ the position for the element e }
Deletion
We visualize \( \text{remove}(p) \), where \( p == \text{last}() \)

Deletion Algorithm

Algorithm \( \text{remove}(p) \):  
\[ t = p.\text{element} \]  
\[ (p.\text{getPrev}).\text{setNext}(p.\text{getNext}) \]  
\[ (p.\text{getNext}).\text{setPrev}(p.\text{getPrev}) \]  
Disallocate node \( p \)  
\[ \text{return } t \]  

Worst-case running time
In a doubly linked list
  + insertion at head or tail is in \( \mathcal{O}(1) \)
  + deletion at either end is on \( \mathcal{O}(1) \)
  -- element access is still in \( \mathcal{O}(n) \)