Algorithms and Data Structures

Chapter 10

Catherine Durso

cdurso@cs.du.edu

Chapter 10

A dynamic set, that is, a set that can have elements added or deleted, is a fundamental abstract data type. A data structure implementing a dynamic set will typically support queries, SEARCH(S, k), insertion, INSERT(S, x), and deletion, DELETE(S, x). Additional operations may be available.

Chapter 10 presents four basic data structures: stacks, queues, linked lists, and rooted trees. These data structures can be used to implement dynamic sets, We will examine the performance of these data structures on the basic operations, and additional operations suited to the data structures.

Stacks

A stack is an implementation of a dynamic set in which the element removed from the set must be the one most recently added (LIFO). The basic operations are PUSH(S, x) (the stack version of INSERT(S, x)) and POP(S), which removes and returns the 'top' of the stack, the value most recently added.

Array-based Stack

A stack can be implemented conveniently with an array.
The stack consists of the elements $S[1..S.top]$.

STACK-EMPTY(S)
1. if $S.top == 0$
2. return TRUE
3. else return FALSE

cont.

PUSH(S, x)

1. $S.top = S.top + 1$
2. $S[S.top] = x$

POP(S)

1. if STACK-EMPTY(S)
2. error underflow
3. else $S.top = S.top - 1$
4. return $S[S.top + 1]$

Asymptotic bounds?

Some Applications

Stacks are used to keep track of function calls, to check syntax, in evaluating postfix expressions, and in converting expressions in standard algebraic notation, called infix, to postfix, a form more easily evaluated by the computer.

Postfix example: $5, 3, +, 10, *$ is evaluated reading from left to right. Operands are stacked. When an operator is read, the appropriate number of operands are popped, the operator applied, and the result pushed. $[5]$ $[5, 3]$ $[8]$ $[8, 10]$ $[80]$

Infix to Postfix

The conversion from infix to postfix uses two stacks, an expression stack and an operator stack. The expression is read from left to right. operands are pushed onto the expression stack.

Operators * and / have higher priority than + and -, which have higher priority than '('. When an operator *,/,+, or - is read, the operator stack is popped and the result pushed onto the expression stack until the top of the operator stack is of lower priority that the operator just read. Then that operator is pushed onto the operator stack.

When a '(' is encountered, it is pushed onto the operator stack. When a ')' is read, the operator stack is popped and the result pushed onto the expression stack until the '(' is encountered. It is popped and discarded.

At the end of the expression, pop and push all remaining operators in the operator stack.

Example

$(10-2)/8$:
expression:$[10, 2]$
operator:$[(, -]$

expression:$[10, 2, -, 8]$
operator:$[/]$

expression:$[10, 2, -, 8, /]$
operator:$[]$

Queues

A queue is an implementation of a dynamic set in which elements are added to the 'back' and removed from the 'front' (FIFO). The basic operations are ENQUEUE(S, x) (the queue version of INSERT(S, x)) and DEQUEUE(S), which removes and returns the 'front' of the queue, the oldest remaining value.

A queue may be implemented with an array if the array is treated as circular.

Array-based Queue

ENQUEUE(Q, x)

1. $Q[Q.tail] = x$
2. if $Q.tail == Q.length$
3. $\quad Q.tail = 1$
4. else $Q.tail = Q.tail + 1$

DEQUEUE(Q)

1. $x = Q[Q.head]$
2. if $Q.head == Q.length$
3. $\quad Q.head = 1$
4. else $Q.head = Q.head + 1$
5. return x

Initially, $Q.head = Q.tail = 1$. A more robust implementation would detect overflow and underflow.

Asymptotic performance?

Linked List

A linked list is a data structure in which the elements are arranged in linear order, but that order is maintained by pointers, rather than by position in an array.

The list has a single data member, the list element $head$ at the front of the list. An empty list $head =$ NIL.

List Varieties

In a singly linked list, each element x consists of a key $x.key$ and a pointer $x.next$ to the next element in the list. The final element in the list has a NIL pointer as $x.next$.

In a doubly linked list, each element x consists of a key $x.key$, a pointer $x.next$ to the next element in the list, and a pointer $x.prev$ to the previous element in the list. The final element in the list has a NIL pointer as $x.next$. The first element in the list has a NIL pointer as $x.prev$.

List Search

LIST-SEARCH(L, k)

1. $x = L.head$
2. while $x \neq$ NIL and $x.key \neq k$
3. $\quad x = x.next$
4. return x

This works for doubly linked lists and for singly linked lists. What is its worst case running time?

List Insertion

LIST-INSERT(L, x)

1. $x.next = L.head$
2. if $L.head \neq$ NIL
3. $L.head.prev = x$
4. $L.head = x$
5. $x.prev =$ NIL

How should this be altered for singly linked lists?. What is its worst case running time?

List Deletion

LIST-DELETE(L, x)

1. if $x.prev \neq$ NIL
2. $\quad x.prev.next = x.next$
3. else $L.head = x.next$
4. if $x.next \neq$ NIL
5. $\quad x.next.prev = x.prev$

How should this be altered for singly linked lists (trick question)? What is its worst case running time?

Sentinel Links

Alternatively, a list may be implemented with an empty sentinel link, $L.nil$ at the head. At the end of the list, a pointer to $L.nil$ is used instead of a NIL pointer. Also, $L.nil.next$ points to the first element in the list. Thus simplifies the insertion and deletion code, but requires extra space. Search is essentially unchanged.

LIST-SEARCH'(L, k)

1. $x = L.nil.next$
2. while $x \neq L.nil$ and $x.key \neq k$
3. $\quad x = x.next$
4. return x

Mutators with Sentinel

LIST-DELETE'(L, x)

1. $x.prev.next = x.next$
2. $x.next.prev = x.prev$

LIST-INSERT'(L, x)

1. $x.next = L.nil.next$
2. $L.nil.next.prev = x$
3. $L.nil.next = x$
4. $x.prev = L.nil$

Issues

Note that the deletion assumes you have the element to delete, x, not just the value.

Insertion and deletion after a particular list element x can also be done. For a singly linked list, insertion or deletion in front of an element x requires finding the previous element, $\Theta(n)$ worst case, unless you do a 'lazy' version, which leaves the desired values in the list, but may make element variables obsolete.

Implementation

In Java, link nodes for a doubly linked list will have the value stored in the link node and the link nodes $next$ and $prev$ as data members.

In C or C++, the link node will have value stored in the link node and the link node pointers $next$ and $prev$ as data members.

To implement singly linked lists in lower level languages, you can use two arrays, one for values and one for the indices of the next value.

List with Arrays

Declare two arrays of size n, one for values, K and one for $next$ indices, N. Declare index variables $head$ and $free$. Initialize $N = [2, 3, ...n, NIL]$, $free = 1$, and $head = NIL$.

When the list is up and running, if ith position is occupied, $K[i]$ will hold a value and $N[i]$ is the index of the next list element.

Insert

INSERT-HEAD $(K, N, head, free, k)$

1. if $free == NIL$
2. error 'out of space'
3. $x = free$ /pop x from the free stack
4. $free = N[free]$
5. $K[x] = k$ /load the new node at x
6. $N[x] = head$
7. $head = x$

Delete

DELETE-NEXT $(K, N, head, free, x)$

1. if $N(x) == NIL$
2. error 'NIL delete'
3. $y = N(x)$
4. $N[x] = N[y]$ /update 'next' for x
5. $N[y] = free$ /push y onto the free stack
6. $free = y$

Doubly Linked

Your text implements a doubly linked list with three arrays. The pseudocode provided is for allocating and freeing list nodes. These would be called by insertion and deletion routines.

Rooted Trees

For a binary tree, each tree node will typically have left, right, and parent nodes or pointers, as well as a value. The tree will just store the root node, or a pointer to the root.

A tree with unbounded branching can be represented using nodes that have parent, left child, and right sibling pointers.

Tree Operations

Breadth-first search can be accomplished using a queue: check the root, and queue all children of the root. While the queue is not empty, dequeue and check a node, then enqueue its children.

Arrays?

How do stacks, queues, and lists improve on arrays? A major liability of arrays is that insertion and deletion may necessitate rewriting a substantial portion of the data, making these worst-case $\Theta(n)$ operations.

Stacks and queues simplify achieving $\Theta(1)$ performance by for insertion and deletion by allowing only restricted deletions. Lists allow for general insertions and deletions.

These structures do not facilitate time-efficient searches.