**COMP1927: B: Sets, Lists, Stacks, Queues**

|  |  |
| --- | --- |
| Simple Collection Types |  |

Collections are a group of data items. Fundamental collection types include:

* *Set* ... unordered collection of distinct items
* *List* ... sequence of items
* *Stack* ... LIFO (Last In First Out) list of items
* *Queue* … FIFO (First In First Out) list of items

For each type, implemented as an ADT:

* Consider interface + alternative implementations
* Analyse performance of each implementation

|  |  |
| --- | --- |
| Sets |  |

|  |  |
| --- | --- |
| **Set ADT** |  |

Book-keeping operations:

* **Set newSet()** ... create new empty set
* **void dropSet(Set)** ... free memory used by set
* **void showSet(Set)**... display / show set as {1,2,3...}

Assignment operations:

* **void readSet(Set)**... read + insert set values
* **Set SetCopy(Set)** ... make a copy of a set

Data-type operations:

* **void SetInsert(Set,Item)**... add item into set
* **void SetDelete(Set,Item)** ... remove item from set
* **int SetMember(Set,Item)**... set membership test
* **Set SetUnion(Set,Set)**... union / combines two sets (returns newly-created set)
* **Set SetIntersect(Set,Set)**... intersection (returns newly-created set)
* **int SetCard(Set)**... cardinality / # of elements

Iterator operations:

* **void SetScanStart(Set)**... set current to first
* **Item SetScanNext(Set)**... fetch next element
* **int SetScanDone(Set)**... reached end of scan?

Note that we are implementing a GADT (Generic Abstract Data Type):

define Item type in separate Item.h

define core operations on Items here as well

Example set client:

int main(int argc, char \*argv[])

{

Set s1, s2, s3;

... put values into Sets ...

printf("s1:"); showSet(s1); printf("\n");

printf("s2:"); showSet(s2); printf("\n");

printf("Cardinality(s2): %d\n", SetCard(s2));

s3 = SetUnion(s1,s2);

printf("Union:"); showSet(s3); printf("\n");

SetFree(s3); // avoid memory leak

s3 = SetIntersect(s2,s1);

printf("Intersect:"); showSet(s3); printf("\n");

return 0;

}

|  |  |
| --- | --- |
| ... Set ADT | 8/92 |

Creating a set of small odd numbers:

Set s; int i;

s = newSet();

for (i = 1; i < 26; i += 2)

SetInsert(s,i);

showSet(s); putchar('\n');

Outputs:

{1,3,5,7,9,11,13,15,17,19,21,23,25}

|  |  |
| --- | --- |
| ... Set ADT | 9/92 |

Iterating over Sets (analogous to files):

Set s; Item it;

...

SetScanStart(s);

while (!SetScanDone(s)) {

it = SetScanNext(s);

... *process item* it ...

}

Note: iteration is incorporated into Set data structure

|  |  |
| --- | --- |
| Set Applications | 10/92 |

Example: eliminating duplicates

// scan a list of items in a file

Set seenItems = newSet();

FILE \*in = fopen(*FileName*,"r");

// getItem() returns false at EOF

// when not EOF, sets value of item

while (getItem(in, &item)) {

if (SetMember(seenItems, item))

// ignore, already processed;

else

*process* item;

}

fclose(in);

|  |  |
| --- | --- |
| ... Set Applications | 11/92 |

Example: counting distinct items

// char \*files[] is a list of file names

// each file contains a list of items

Set seenItems = newSet();

for (i = 0; i < NFILES; i++) {

FILE \*in = fopen(files[i],"r");

while (getItem(in,&item)) {

*process* item;

SetInsert(seenItems, item);

}

fclose(in);

}

int nitems = SetCard(seenItems);

printf("Got %d distinct items\n", nitems);

|  |  |
| --- | --- |
| Set Implementation | 12/92 |

Implement ADTs for Sets of int values ...

using an unsorted array

using a sorted array

using a linked-list

using a hash-table

using a bit-map

Compare and contrast efficiency of operations.

|  |  |
| --- | --- |
| Sets as Unsorted Arrays | 13/92 |

Concrete representation:

#define MAXSET 1000

typedef struct setRep {

int nelems;

int curr;

Item values[MAXSET];

} SetRep;

Set newSet()

{

Set s;

s = malloc(sizeof(setRep));

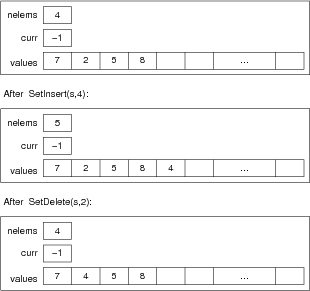
assert(s != NULL);

s->nelems = 0; s->curr = -1;

return s;

}

|  |  |
| --- | --- |
| ... Sets as Unsorted Arrays | 14/92 |



|  |  |
| --- | --- |
| ... Sets as Unsorted Arrays | 15/92 |

Set operation implementations (with error-checking removed)

void SetInsert(Set s, Item it)

{

s->vals[s->nelems] = ItemCopy(it);

s->nelems++;

}

int SetCard(Set s)

{

return s->nelems;

}

int SetMember(Set s, Item it)

{

int i;

for (i = 0; i < s->nelems; i++)

if (ItemEQ(s->vals[i], it)) return 1;

return 0;

}

Set SetUnion(Set s1, Set s2)

{

Set new = SetCopy(s1);

int i;

for (i = 0; i < s2->nelems; i++)

SetInsert(new, s2->vals[i]);

return new;

}

Set SetIntersect(Set s1, Set s2)

{

Set new = newSet();

int i;

for (i = 0; i < s1->nelems; i++) {

if (SetMember(s2, s1->vals[i]))

SetInsert(new, s1->vals[i]);

}

return new;

}

|  |  |
| --- | --- |
| Sets as Unsorted Arrays | 16/92 |

Costs for set operations on unsorted array:

*card*: read from struct;   constant cost   *O(1)*

*member*: scan list from start;   linear cost   *O(n)*

*insert*: duplicate check, add at end;   linear cost   *O(n)*

*delete*: find, copy last into gap;   linear cost   *O(n)*

*union*: copy s1, insert each item from s2;   quadratic cost   *O(nm)*

*intersect*: scan for each item in s1;   quadratic cost   *O(nm)*

Assuming: s1 has *n* items, s2 has *m* items

|  |  |
| --- | --- |
| ... Sets as Unsorted Arrays | 17/92 |

Analysis for Union

Res = SetCopy(s1)

foreach item in s2 { Res = SetInsert(Res,item) }

Union = copy cost + m.(insert cost) = n + n.m ~ *O(nm)*

Analysis for Intersect

Res = empty

for each item in s1

{ if (item ∈ s2) SetInsert(Res,item) }

Intersect = scan cost + n.(member check) = n + n.m ~ *O(nm)*

|  |  |
| --- | --- |
| Sets as Sorted Arrays | 18/92 |

Same data structure as for unsorted arrays.

Differences:

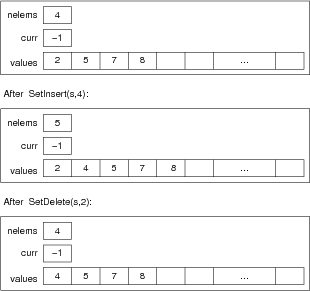
*member* can use binary search

*insert* requires: find location, shift, insert

*delete* requires: find location, shift down

*union* and *intersect* could use merging

|  |  |
| --- | --- |
| ... Sets as Sorted Arrays | 19/92 |



|  |  |
| --- | --- |
| ... Sets as Sorted Arrays | 20/92 |

Costs for set operations on sorted array:

*card*: read from struct;   *O(1)*

*member*: binary search;   *O(log n)*

*insert*: find, shift up, insert;   *O(n)*

*delete*: find, shift down;   *O(n)*

*union*: merge = scan s1, scan s2;   *O(n)*

*intersect*: merge = scan s1, scan s2;   *O(n)*

|  |  |
| --- | --- |
| Sets as Linked-lists | 21/92 |

Concrete representation:

typedef struct Node {

Item data;

Node \*next;

} Node;

typedef struct setRep {

int nelems;

Node \*curr;

Node \*list;

} SetRep;

Set newSet()

{

Set s;

s = malloc(sizeof(setRep));

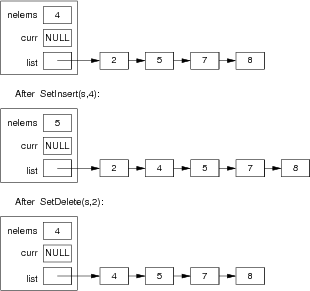
s->nelems = 0; s->curr = NULL;

s->list = NULL;

return s;

}

|  |  |
| --- | --- |
| ... Sets as Linked-lists | 22/92 |



|  |  |
| --- | --- |
| ... Sets as Linked-lists | 23/92 |

Costs for set operations on sorted list:

*insert*: find, link in;   *O(n)*

*delete*: find, unlink;   *O(n)*

*member*: linear search;   *O(n)*

*card*: lookup;   *O(1)*

*union*: merge = scan s1, scan s2;   *O(n)*

*intersect*: merge = scan s1, scan s2;   *O(n)*

For *member*, on average scan only half of list (sorted)

|  |  |
| --- | --- |
| ... Sets as Linked-lists | 24/92 |

Costs for set operations on unsorted list:

*insert*: duplicate check, insert at head;   *O(n)*

*delete*: find, unlink;   *O(n)*

*member*: linear search; *O(n)*

*card*: lookup; *O(1)*

*union*: copy s1, insert each item from s2; *O(nm)*

*intersect*: scan for each item in s1;   *O(nm)*

Assume *n* = size of *s1*, *m* = size of *s2*

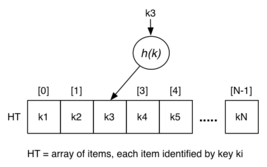
If we drop nelems, *card* becomes *O(n)*

|  |  |
| --- | --- |
| Sets as Hash Tables | 25/92 |

A *hash table* is a data structure that

provides fast access to items

based on a *key* value



|  |  |
| --- | --- |
| ... Sets as Hash Tables | 26/92 |

Hash function *h(k)* has type *KeyDomain* → *0..N-1*

Converts *key* value *k* into array index.

Ideally, to implement a Set

define a hash table of size *N*

each entry in the table contains an item or is empty

take item value, hash to location

if hash table entry is empty, item is not in set

|  |  |
| --- | --- |
| ... Sets as Hash Tables | 27/92 |

If hash table is perfect

each item value maps to a different location

checking for a value is constant time operation (*O(1)*)

But, we have a problem ...

number of keys is much larger than *N*

inevitably, different keys produce same location

Known as a *hash collision*.

|  |  |
| --- | --- |
| ... Sets as Hash Tables | 28/92 |

For the Set ADT, use a simple form of collision handling:

have an array of *N* linked lists

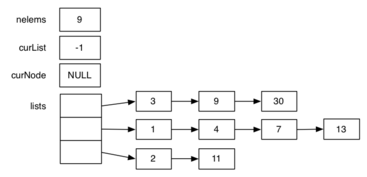
take (value % N) as index i (i.e. as hash function)

manipulate just the ith list

each list will (hopefully) be around the size *n/N*

Making *N* larger makes lists shorter ⇒ faster to check

|  |  |
| --- | --- |
| ... Sets as Hash Tables | 29/92 |



Hash function is (value % 3).

*insert*: compute which list; find location; link in

*delete*: compute which list; find location; unlink

*member*: compute which list; scan to find

|  |  |
| --- | --- |
| ... Sets as Hash Tables | 30/92 |

Concrete representation:

#define MAXLISTS ??

typedef struct Node {

Item data;

Node \*next;

} Node;

typedef struct SetRep {

int nelems;

int curList;

Node \*curNode;

Node \*lists[MAXLISTS];

} SetRep;

Set newSet()

{

Set s; int i;

s = malloc(sizeof (struct setRep));

s->nelems = 0;

s->curList = -1; s->currNode = NULL;

for (i = 0; i < MAXLISTS; i++)

s->lists[i] = NULL;

return s;

}

|  |  |
| --- | --- |
| ... Sets as Hash Tables | 31/92 |

Costs for set operations on simple hash table:

*insert*: find, link in;   *O(n)*

*delete*: find, unlink;   *O(n)*

*member*: linear search;   *O(n)*

*card*: lookup;   *O(1)*

*union*: merge = scan s1, scan s2;   *O(n)*

*intersect*: merge = scan s1, scan s2;   *O(n)*

Note: same *O(...)* as for linked lists.

What changed? *Actual* cost reduced by *1/MAXLISTS*

|  |  |
| --- | --- |
| Sets as Bit-maps | 32/92 |

Restrict possible values that can be stored in the Set

represent each value by position in large array of bits

insertion means set a bit to 1 (bit|1)

deletion means set a bit to 0 (bit&0)

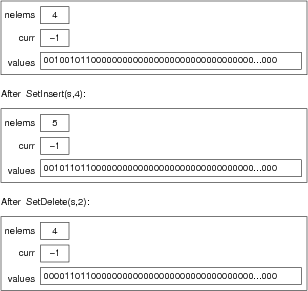
bit position for value i is easy to compute

Representation of bit-list in C:

#define NBITS = 32\*NWORDS

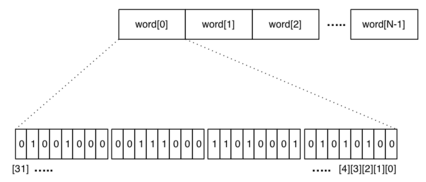
unsigned int bits[NWORDS];

|  |  |
| --- | --- |
| ... Sets as Bit-maps | 33/92 |



|  |  |
| --- | --- |
| ... Sets as Bit-maps | 34/92 |

Details of long bit-string:



|  |  |
| --- | --- |
| ... Sets as Bit-maps | 35/92 |

Implementing large bit-maps in C

done as array of unsigned ints (words)

each unsigned int is (typically) 32 bits long

array of length N words can represent 0..32\*N-1

to update bit i

determine which array element (i/32)

determine which bit position in word (i%32)

|  |  |
| --- | --- |
| ... Sets as Bit-maps | 36/92 |

#define NWORDS ??

#define NBITS (32\*NWORDS)

typedef unsigned int bits[NWORDS] Bits;

setBit(Bits b, int i)

{

int wordIndex, bitIndex;

unsigned int mask;

assert(i < NBITS);

wordIndex = i/32;

bitIndex = i%32;

mask = (1 < bitIndex);

b[wordIndex] = b[wordIndex] | mask;

}

unsetBit(Bits b, int i)

{

int wordIndex, bitIndex;

unsigned int mask;

assert(i < NBITS);

wordIndex = i/32;

bitIndex = i%32;

mask = ~(1 < bitIndex);

b[wordIndex] = b[wordIndex] & mask;

}

|  |  |
| --- | --- |
| ... Sets as Bit-maps | 37/92 |

Costs for set operations on bit-string:

*insert*: compute position, set bit;   *O(1)*

*delete*: compute position, set bit;   *O(1)*

*member*: compute position, check bit;   *O(1)*

*card*: lookup;   *O(1)*

*union*: scan s1, scan s2, OR words;   *O(n)*

*intersect*: scan s1, scan s2, AND words;   *O(n)*

|  |  |
| --- | --- |
| Set Implementations | 38/92 |

Performance comparison (*n = #elems, N = max elems*):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Representation | insert | delete | member | card | storage |
| unsorted array | *O(n)* | *O(n)* | *O(n)* | *O(1)* | *O(N)* |
| sorted array | *O(n)* | *O(n)* | *O(log2n)* | *O(1)* | *O(N)* |
| linked list | *O(n)* | *O(n)* | *O(n)* | *O(1)* | *O(n)* |
| hashed lists | *O(n)* | *O(n)* | *O(n)* | *O(1)* | *O(n)* |
| bit-maps | *O(1)* | *O(1)* | *O(1)* | *O(1)* | *O(N)* |

For card, assume we keep a count of elems.   
Bit-map version restricts elements to integers *0..N*

|  |  |
| --- | --- |
| Lists |  |

|  |  |
| --- | --- |
| List ADT | 40/92 |

List data type: sequence of items (accessed by position)

have identifiable first/second/third/... items

Variations:

item values: sorted, *not sorted*

duplicate values: *allowed*, not allowed

items inserted: at start, *at end*, in order

items deleted: *by value*, by position

For ordered lists: Item type must have GT operation

|  |  |
| --- | --- |
| ... List ADT | 41/92 |

Book-keeping operations:

List newList() ... create new empty list

void dropList(List) ... free memory used by list

void showList(List) ... display as [1,2,3...]

Assignment operations:

void readList(List) ... read+insert list values

List ListCopy(List) ... make a copy of a list

|  |  |
| --- | --- |
| ... List ADT | 42/92 |

Data-type operations:

void ListInsert(List,Item) ... add item into list

void ListDelete(List,Item) ... remove item(s)

Item ListItem(List,int) ... get i'th item

int ListFindPos(List,Item) ... position of item

int ListLength(List) ... # items in list

void ListStartScan(List) ... begin scan

Item ListScanNext(List) ... fetch next element

int ListScanDone(List) ... reached end of scan?

Note: ListFindPos() returns distinguished value if not in list

|  |  |
| --- | --- |
| ... List ADT | 43/92 |

Sedgewick defines a lower-level List ADT which

deals explicitly with Nodes as well as Items

could be used to implement most linear data types

Example interface:

typedef struct ListNode \*Node;

Node newNode(Item);

void freeNode(Node);

void insertNext(Node,Node);

Node deleteNext(Node);

Node next(Node);

Item item(Node);

|  |  |
| --- | --- |
| Applications of Lists | 44/92 |

Lists are used frequently, in many applications.

Typically, each list is specialised for task at hand.

E.g. unsorted, duplicates, insert at end, delete by key, ...

Lists form the basis for many other data types.

|  |  |
| --- | --- |
| ... Applications of Lists | 45/92 |

Simple client, for testing:

int main(int argc, char \*argv)

{

List L = newList();

for (i = 0; i < 20; i++)

ListInsert(L,i);

assert(ListLength(L) == 20);

printf("L: "); showList(L); printf("\n");

int curr, prev = -1;

ListStartScan(L);

while (!ListScanDone(L)) {

curr = ListScanNext(L);

assert(curr > prev);

prev = curr;

}

dropList(L);

}

|  |  |
| --- | --- |
| Lists as Arrays | 46/92 |

Concrete representation:

typedef struct ListRep {

int size; // capacity of array

int last; // index of last elem

int curr; // index for scannning

Item \*values; // array of values

} ListRep;

List newList()

{

List L;

L = malloc(sizeof(ListRep));

assert(L != NULL);

L->values = malloc(100\*sizeof(Item));

assert(L->values != NULL);

L->size = 100;

L->last = -1;

L->curr = 0;

return L;

}

void ListInsert(List L, Item it)

{

if (L->last+1 == L->size)

... increase array size ...

L->last++;

L->values[L->last] = ItemCopy(it);

}

|  |  |
| --- | --- |
| ... Lists as Arrays | 47/92 |

Cost for List operations:

*copy*: scan source, insert each in target;  *O(n)*

*insert*: add after last item;  *O(1)*\*

*delete*: scan, remove item occurences;  *O(n)*

*item*: index into array;  *O(1)*

*find*: scan until found;  *O(n)*

*length*: read from last;  *O(1)*

\*Occasional array expansion cost is *O(n)*

|  |  |
| --- | --- |
| Lists as Linked Lists | 48/92 |

Concrete representation:

typedef struct ListNode {

Item value;

struct ListNode \*next;

} ListNode;

typedef struct ListRep {

ListNode \*first; // ptr to first node

ListNode \*last; // ptr to last node

ListNode \*curr; // current node in scan

} ListRep;

List newList()

{

List L;

L = malloc(sizeof(ListRep));

assert(L != NULL);

L->first = NULL;

L->last = NULL;

L->curr = NULL;

return L;

}

void ListInsert(List L, Item it)

{

ListNode \*new = malloc(sizeof(ListNode));

assert(new != NULL);

new->value = ItemCopy(it);

new->next = NULL;

if (L->first == NULL) L->first = new;

if (L->last != NULL) L->last->next = new;

L->last = new;

}

|  |  |
| --- | --- |
| ... Lists as Linked Lists | 49/92 |

Cost for List operations:

*copy*: scan source, insert each in target;  *O(n)*

*insert*: add after last node;  *O(1)*

*delete*: scan, remove item occurences;  *O(n)*

*item*: scan to i'th node;  *O(n)*

*find*: scan until found;  *O(n)*

*length*: scan and count;  *O(n)*\*

If ListRep also contains #nodes, then *length* has cost *O(1)*

|  |  |
| --- | --- |
| Stacks |  |

|  |  |
| --- | --- |
| Stack ADT | 51/92 |

Stack data type: LIFO list of items

Characterised by get / push operations

items always inserted at top of stack

items always removed from top of stack

A specialisation of a List where

not sorted, duplicates allowed

insert at top, delete from top

|  |  |
| --- | --- |
| ... Stack ADT | 52/92 |

Book-keeping operations:

Stack newStack() ... create new empty stack

void dropStack(Stack) ... free memory used by stack

showStack(Stack) ... display as 3 on 5 on 4 on ...

Data-type operations:

void StackPush(Stack,Item) ... add item on stack

Item StackPop(Stack) ... remove item from stack

int StackIsEmpty(Stack) ... check for no items

|  |  |
| --- | --- |
| Stacks as Arrays | 53/92 |

Concrete representation:

typedef struct StackRep {

int size; // capacity of array

int top; // index of last elem

Item \*values; // array of values

} StackRep;

Stack newStack()

{

Stack s;

s = malloc(sizeof(StackRep));

assert(s != NULL);

s->values = malloc(100\*sizeof(Item));

assert(s->values != NULL);

s->size = 100;

s->last = -1;

s->curr = 0;

return s;

}

void StackPush(Stack s, Item it)

{

if (s->top+1 == s->size)

... increase array size ...

s->top++;

s->values[s->top] = ItemCopy(it);

}

|  |  |
| --- | --- |
| ... Stacks as Arrays | 54/92 |

Cost for Stack operations:

*push*: add after last item;  *O(1)*

*leave*: decrement top index;  *O(1)*

*empty?*: return top index+1;  *O(1)*

|  |  |
| --- | --- |
| Stacks as Linked Lists | 55/92 |

Concrete representation:

typedef struct StackNode {

Item value;

struct StackNode \*next;

} StackNode;

typedef struct StackRep {

StackNode \*top; // ptr to first node

} StackRep;

Stack newStack()

{

Stack s;

s = malloc(sizeof(StackRep));

assert(s != NULL);

s->top = NULL;

return s;

}

void StackPush(Stack s, Item it)

{

StackNode \*new = malloc(sizeof(StackNode));

assert(new != NULL);

new->value = ItemCopy(it);

new->next = s->top;

s->top = new;

}

|  |  |
| --- | --- |
| ... Stacks as Linked Lists | 56/92 |

Cost for Stack operations:

*push*: add as first node;  *O(1)*

*pop*: remove first node;  *O(1)*

*empty?*: check for empty list;  *O(1)*

|  |  |
| --- | --- |
| Queues |  |

|  |  |
| --- | --- |
| Queue ADT | 58/92 |

Queue data type: FIFO list of items

Characterised by enqueue / dequeue operations

items always join at end of queue

items always leave from front of queue

A specialisation of a List where

not sorted, duplicates allowed

insert at end, remove from start

|  |  |
| --- | --- |
| ... Queue ADT | 59/92 |

Book-keeping operations:

Queue newQueue() ... create new empty queue

void dropQueue(Queue) ... free memory used by queue

showQueue(Queue) ... display as 3 > 5 > 4 on ...

Data-type operations:

void QueueJoin(Queue,Item) ... add item on queue

Item QueueLeave(Queue) ... remove item from queue

int QueueIsEmpty(Queue) ... check for no items

|  |  |
| --- | --- |
| ... Queue ADT | 60/92 |

Simple Queue application:

#define NQUEUES 10

int main(int argc, char\* argv[])

{

int i, j, N, nQ = NQUEUES;

N = (argc < 2) ? 100 : atoi(argv[1]);

Queue q[NQUEUES];

for (i = 0; i < nQ; i++)

q[i] = newQueue();

for (j = 0; j < N; j++)

QueueJoin(q[rand()%nQ], j);

for (i = 0; i < N; i++) {

showQueue(q[i]); printf("\n");

dropQueue(q[i]);

}

}

Applications: Queues are frequently used in discrete event simulation.

|  |  |
| --- | --- |
| Queues as Arrays | 61/92 |

Concrete representation:

typedef struct QueueRep {

int size; // capacity of array

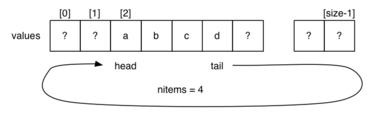
int head; // index of front elem

int tail; // index of back elem

int nitems; // # items in queue

Item \*values; // array of values

} QueueRep;



|  |  |
| --- | --- |
| ... Queues as Arrays | 62/92 |

Queue newQueue()

{

Queue q;

q = malloc(sizeof(QueueRep));

assert(q != NULL);

q->values = malloc(100\*sizeof(Item));

assert(q->values != NULL);

q->size = 100;

q->head = -1;

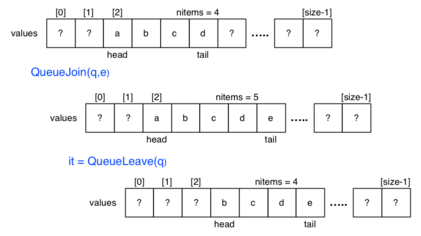
q->tail = -1;

q->nitems = 0;

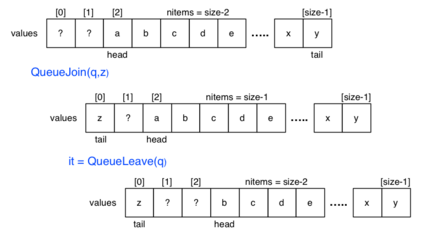
return q;

}

|  |  |
| --- | --- |
| ... Queues as Arrays | 63/92 |



|  |  |
| --- | --- |
| ... Queues as Arrays | 64/92 |



|  |  |
| --- | --- |
| ... Queues as Arrays | 65/92 |

void QueueJoin(Queue q, Item it)

{

if (q->nitems == q->size)

... increase array size ...

q->tail = (q->tail + 1) % q->size;

q->values[q->tail] = ItemCopy(it);

q->nitems++;

}

Item QueueLeave(Queue q)

{

assert(q->nitems > 0);

Item it = ItemCopy(q->values[q->head]);

q->head = (q->head + 1) % q->size;

q->nitems--;

return it;

}

|  |  |
| --- | --- |
| ... Queues as Arrays | 66/92 |

Cost for Queue operations:

*join*: add after tail;  *O(1)*

*leave*: increment head index;  *O(1)*

*empty?*: check nelems;  *O(1)*

|  |  |
| --- | --- |
| Queues as Linked Lists | 67/92 |

Concrete representation:

typedef struct QueueNode {

Item value;

struct QueueNode \*next;

} QueueNode;

typedef struct QueueRep {

QueueNode \*head; // ptr to first node

QueueNode \*tail; // ptr to last node

} QueueRep;

Queue newQueue()

{

Queue q;

q = malloc(sizeof(QueueRep));

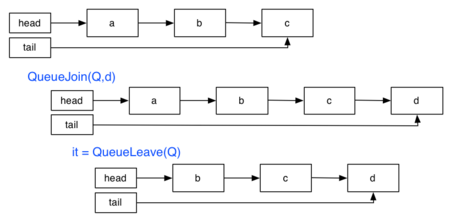
assert(q != NULL);

q->head = q->tail = NULL;

return q;

}

|  |  |
| --- | --- |
| ... Queues as Linked Lists | 68/92 |



|  |  |
| --- | --- |
| ... Queues as Linked Lists | 69/92 |

void QueueJoin(Queue q, Item it)

{

QueueNode \*new = malloc(sizeof(QueueNode));

assert(new != NULL);

new->value = ItemCopy(it);

new->next = NULL;

if (q->head == NULL)

q->head = new;

if (q->tail != NULL)

q->tail->next = new;

q->tail = new;

}

Item QueueLeave(Queue q)

{

assert(q->head != NULL);

Item it = ItemCopy(q->head->value);

QueueNode \*old = q->head;

q->head = q->head->next;

free(old);

return it;

}

|  |  |
| --- | --- |
| ... Queues as Linked Lists | 70/92 |

Cost for Queue operations:

*join*: add as first node;  *O(1)*

*leave*: remove first node;  *O(1)*

*empty?*: check for empty list;  *O(1)*

|  |  |
| --- | --- |
| Priority Queues |  |

|  |  |
| --- | --- |
| Priority Queues | 72/92 |

Some applications of queues require

items processed in order of "key"

rather than in order of entry (i.e. FIFO)

Example applications include:

discrete event simulation (key: event time)

scheduling in operating systems (key: process priority)

graph searching (key: distance to destination)

artificial intelligence (key: optimizing cost function)

|  |  |
| --- | --- |
| ... Priority Queues | 73/92 |

Characteristic priority queue (PQ) operations:

insert item

remove item with largest key

Along with generic ADT operations, such as:

create new queue

test if queue is empty

destroy existing queue (reclaim memory)

|  |  |
| --- | --- |
| ... Priority Queues | 74/92 |

Example of use:

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Insert | Priority Queue | Remove |
| 0 | - | *Empty* | - |
| 1 | E | E | - |
| 2 | X | E X | - |
| 3 | A | E X A | - |
| 4 | - | E A | X |
| 5 | M | E A M | - |
| 6 | - | E A | M |
| 7 | P | E A P | - |
| 8 | L | E A P L | - |
| 9 | - | E A L | P |
| 10 | E | E A L E | - |
| 11 | - | E A E | L |
| 12 | - | E A | E |
| 13 | - | A | E |
| 14 | - | *Empty* | A |

|  |  |
| --- | --- |
| ... Priority Queues | 75/92 |

Priority Queue interface (for real applications):

// Types

... PriQ ... //priority queue

... Item ... //items in queue

... Key ... //priority values in items

// Core operations

PriQ initPriQ(int size);

void insert(PriQ q, Item it);

Item delMax(PriQ q);

// Useful operations

int isEmpty(PriQ q);

void change(PriQ q, Key k, Item it);

void delete(PriQ q, Key k);

PriQ join(PriQ q1, PriQ q2);

|  |  |
| --- | --- |
| Exercise: Stacks and FIFO Queues | 76/92 |

Priority queues are a generalisation of Stack/Queue.

implement a Stack via a priority queue

implement a FIFO Queue via a priority queue

// use just these PQ operations

PriQ initPriQ(int size);

int isEmpty(PriQ q);

void insert(PriQ q, Item it);

Item delMax(PriQ q);

|  |  |
| --- | --- |
| Priority Queue as Array | 77/92 |

Implementation based on unordered arrays:

typedef struct PQrep \*PriQ;

struct PQrep {

int nItems; // count of items

Item \*items; // array of Items

int size; // size of array

}

// create a new empty queue

PriQ initPriQ(int size) {

PriQ q = malloc(sizeof(struct PQrep));

assert(q != NULL);

q->items = malloc(sizeof(Item) \* size);

assert(q->items != NULL);

q->nItems = 0;

q->size = size;

return q;

}

// add a new item into the queue

void insert(PriQ q, Item it) {

assert(q != NULL && q->nItems < q->size);

q->items[q->nItems] = it;

q->nItems++;

}

// delete the largest item

Item delMax(PriQ q) {

assert(q != NULL && q->nItems > 0);

Item \*a = q->items;

int i, max = 0, n = q->nItems-1;

for (i = 1; i <= n; i++)

if (less(a[i] < a[max])) max = i;

swap(a, max, n);

q->nItems = n;

return a[n];

}

|  |  |
| --- | --- |
| ... Priority Queue as Array | 78/92 |

Implementation based on ordered arrays:

// data and initialisation same as above

...

// add a new item into the queue

void insert(PriQ q, Item it) {

assert(q != NULL && q->nItems < q->size);

Item \*a = q->items;

int i, j, n = q->nItems;

for (i = 0; i < n; i++)

if (less(it < a[i])) break;

for (j = n; j > i; j--)

a[j] = a[j-1]; // push up

a[i] = it;

q->nItems++;

}

// delete the largest item

Item delMax(PriQ q) {

q->nItems--;

return q->items[q->nItems];

}

|  |  |
| --- | --- |
| ... Priority Queue as Array | 79/92 |

Example usage of two implementations:

|  |  |  |
| --- | --- | --- |
| Operation | ordered | unordered |
| insert E | E | E |
| insert X | E X | E X |
| insert A | A E X | E X A |
| insert M | A E M X | E X A M |
| delMax | A E M | E A M |
| insert P | A E M P | E A M P |
| insert L | A E L M P | E A M P L |
| delMax | A E L M | E A M L |
| delMax | A E L | E A L |
| insert E | A E E L | E A L E |
| delMax | A E E | E A E |

|  |  |
| --- | --- |
| ... Priority Queue as Array | 80/92 |

Operation costs for PriQ containing *N* items:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Structure | Space | insert | delmax | isEmpty |
| Sorted Array | *MaxN* | *O(N)* | *O(N)* | *O(1)* |
| Unsorted Array | *MaxN* | *O(1)* | *O(N)* | *O(1)* |
| Sorted List | *O(N)* | *O(N)* | *O(1)* | *O(1)* |
| Unsorted List | *O(N)* | *O(1)* | *O(N)* | *O(1)* |

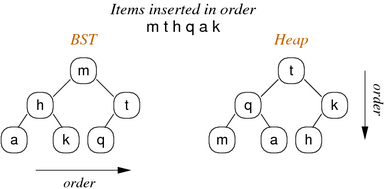
Can we implement *all* operations efficiently?

|  |  |  |
| --- | --- | --- |
| Implementation | insert | delMax |
| heap | *O(logN)* | *O(logN)* |

|  |  |
| --- | --- |
| Heaps |  |

|  |  |
| --- | --- |
| Heaps | 82/92 |

Heaps can be viewed as trees with top-to-bottom ordering   
(cf. binary search trees which have left-to-right ordering)



|  |  |
| --- | --- |
| ... Heaps | 83/92 |

Binary search trees, with root node *k*:

all keys in left subtree *≤ k*

all keys in right subtree *> k*

property applies to all nodes in tree

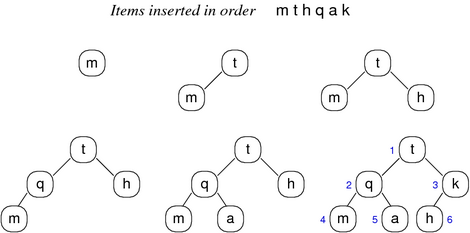
Heap-ordered trees, with root node *k*:

all keys both subtrees are *≤ k*

property applies to all nodes in tree   
(i.e. root contains largest value in that subtree)

|  |  |
| --- | --- |
| ... Heaps | 84/92 |

Heaps grow as follows (level-order):



|  |  |
| --- | --- |
| ... Heaps | 85/92 |

BSTs are typically implemented as linked data structures.

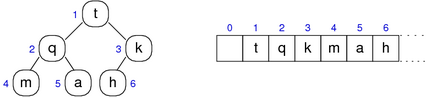
Heaps are typically implemented via arrays.

Simple index calculations allow navigation through the tree:

left child of node at index *i* is located at *2i*

right child of node at index *i* is located at *2i+1*

parent of node at index *i* is located at *i/2*

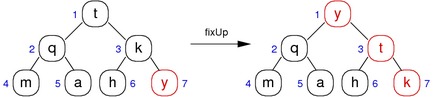


|  |  |
| --- | --- |
| Insertion with Heaps | 86/92 |

Insertion is a two-step process

add new element at bottom-most, rightmost position   
(but this might violate heap property; new value larger than parent)

reorganise values along path to root to restore heap



|  |  |
| --- | --- |
| ... Insertion with Heaps | 87/92 |

Bottom-up heapify:

// force value at a[k] into correct position

void fixUp(Item a[], int k)

{

while (k > 1 && less(a[k/2],a[k])) {

swap(a, k, k/2);

k = k/2; // integer division

}

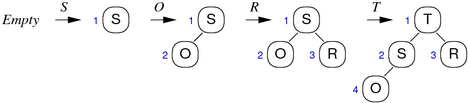
}

|  |  |
| --- | --- |
| Exercise: Heap Construction | 88/92 |

Show the construction of the heap produced by inserting:

S O R T I N G I S F U N

The first four steps:



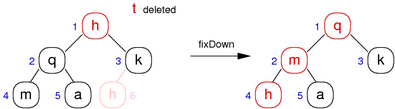
|  |  |
| --- | --- |
| Deletion with Heaps | 89/92 |

Deletion is a three-step process (in PQ context)

replace root value by bottom-most, rightmost value

remove bottom-most, rightmost value

reorganise values along path from root to restore heap



|  |  |
| --- | --- |
| ... Deletion with Heaps | 90/92 |

Top-down heapify:

// force value at a[k] into correct position

void fixDown(Item a[], int k, int N)

{

while (2\*k <= N) {

int j = 2\*k;

// choose larger of two children

if (j < N && less(a[j], a[j+1])) j++;

if (!less(a[k], a[j])) break;

swap(a, k, j);

k = j;

}

}

|  |  |
| --- | --- |
| Priority Queue as Heap | 91/92 |

Implementation based on unordered arrays:

typedef struct PQrep \*PriQ;

struct PQrep {

int nItems; // count of items

Item \*items; // heap-array of Items

int size; // size of array

}

// create a new empty queue

PriQ initPriQ(int size) {

PriQ q = malloc(sizeof(struct PQrep));

assert(q != NULL);

// indexes start from 1

q->items = malloc(sizeof(Item) \* (size+1));

assert(q->items != NULL);

q->nItems = 0;

q->size = size;

return q;

}

// add a new item into the queue

void insert(PriQ q, Item it) {

assert(q != NULL && q->nItems < q->size);

q->nItems++;

q->items[q->nItems] = it;

fixUp(q->items, q->nItems);

}

// delete the largest item

Item delMax(PriQ q) {

assert(q != NULL && q->nItems > 0);

swap(q->items, 1, q->nItems);

q->nItems--;

fixDown(p->items, 1, q->nItems);

return q->items[q->nItems+1];

}

|  |  |
| --- | --- |
| ... Priority Queue as Heap | 92/92 |

Cost of PriQ operations in a heap:

#comparisons/#swaps determined by tree depth

since tree is compact, depth is *log2N*

so, each insert/delete requires *≤ log2N* compares/swaps

Produced: 17 Aug 2014