目录

Priority queues

In a queue:

- items are processed in the order of arrival
- you could say that the first item in the queue always has highest priority

In a priority queue:

- items are assigned *priorities*: they are ordered by priority
- the item with the highest priority (value) is processed first
- the priority may have nothing to do with the order of arrival

A priority queue is a generalization of a stack and a queue

- can implement a stack using a priority queue
- can implement a queue using a priority queue

Like queues and stacks, *priority queues* can be implemented as ADTs and implemented by:

- arrays or
- linked lists

A priority queue supports 2 abstract operations:

- **insert**: a new item (equivalent to *push* for a stack, and *qush* for a queue)
- **delmax**: delete the item with the largest key (equivalent *pop*)

You should also be able to

- create a priority queue
- is the priority queue empty

Inserting and delmaxing

Consider the input sequence *E X A # M # # #*

- where a letter defines an insert operation (push)
  - if implemented as a function, it puts the element in the data structure and returns a *void*
- a # defines a delete operation (pop) and return the value
  - if implemented as a function, it deletes and an element in the data structure and returns the element

If the ADT was a stack, then # deletes the *top* element:

operation	return	stack
insert E	void	E
insert X	void	EX
insert A	void	EXA
pop	A	EX
insert M	void	EXM
pop	M	EX
pop	X	E
pop	E	~

If the ADT was a queue, then # deletes the *bottom* element:

operation	return	queue
insert E	void	E
insert X	void	EX
insert A	void	EXA
pop	E	XA
insert M	void	XAM
pop	X	AM
pop	A	M
pop	M	~

If the ADT is a priority queue then # deletes the *maximum* (or *minimum* element)

- we refer to it as **delmax** (or **delmin**)
- the sequence of operations is:

operation	return	priority queue
insert E	void	E
insert X	void	EX
insert A	void	EXA
delmax	X	EA
insert M	void	EAM
delmax	M	EA
delmax	E	A
delmax	A	~

Notice:

- in the ADT we do not now in what order elements are stored in the abstract data structure
- *delmax* gets the maximum element, but we do not know how *delmax* finds this element

Consider another example: $P\ R\ I\ O\ \#\ R\ \#\ \#\ I\ \#\ T\ \#\ Y\ \#\ \#\ \#\ Q\ U\ E\ \#\ \#\ \#\ U\ \#\ E$

- The order that the elements will be deleted is:
  - first hash deletes the first R
  - second hash deletes the second R
  - third hash deletes P
  - fourth hash deletes O
  - etc

A Priority Queue ADT

Typically an ADT allows the user to:

- **create** an abstract data structure
- **insert** an element into the structure
  - stacks and queues use *push* and *qush*
- **delete** an element from the structure
  - stacks and queues use *pop*
- check whether the structure **is empty**

A generic interface could be:

切换行号显示

```
// pq.h: ADT interface for a priority queue
#include <stdio.h>
#include <stdlib.h>

typedef struct pqRep *PQ;

PQ   createPQ(int size);
void insertPQ(PQ q, int it);
int  delMaxPQ(PQ q);
int  isEmptyPQ(PQ q);
```

Other possible operations in the interface could be:

切换行号显示

```
void  updatePQ(PQ q, int i, int j); // change element value from i to j
void  deletePQ(PQ q);          // remove the PQ completely
PQ    joinPQ(PQ q1, PQ q2);    // concatenate 2 PQs

```

Client: PQ sort

Here is a client of the ADT, called **pqSort()**, that sorts an array of integers.

切换行号显示

```
/* pqSort.c: use a priority queue to sort an array of integers
             into descending order
 */
#include "pq.h"

int main() {
   int a[] = {41, 2, 58, 156, 360, 81, 260, 74, 167, 13};
   int length = sizeof(a)/sizeof(a[0]);

   PQ q = createPQ(length);
   printf("Array: ");
   for (int i = 0; i < length; i++) {
       printf("%d ", a[i]);
       insertPQ(q, a[i]);
   }
   printf("\nSorted: ");
   while (!isEmptyPQ(q)) {
       printf("%d ", delMaxPQ(q));
   }
   putchar('\n');
   return EXIT_SUCCESS;
}
```

This program:

- inserts all the items into the priority queue
- delMaxs items from the priority queue in descending order

If you want the minimum element, rewrite *delmaxPQ()* into *delMinPQ()*

Is the program fast?

- that depends on how *insertPQ()* and *delMaxPQ()* are implemented
- *PQ* is an ADT, so the client does not know

Is the program efficient in space used?

- *No*, because
    - each item in the array is inserted into the priority queue
        - ... which is also an array, so 2 arrays are being used!
- it would be better if *pqSort()* was *in-place*

Three implementations of the priority queue ADT

We can implement the interface functions using different concrete data structures:

1. an unordered array
    - items are simply appended to the array as they are entered
2. an ordered array
    - items are inserted into their correct position in an array
3. a heapified array (i.e. a heap)
    - items are added to the array that is subsequently heapified

Note, all implementations use arrays, but they differ in how/where they store the items in the array.

PQ ADT implementation using an unordered array

切换行号显示

```c
// pqUA.c: priority queue implementation for pq.h using an unordered array
#include "pq.h"

struct pqRep {
   int nItems;  // actual count of Items
   int *items;  // array of Items
   int maxsize; // maximum size of array
};

PQ createPQ(int size) {
   PQ q = malloc(sizeof(struct pqRep));  // make room for the structure
   if (q == NULL) {
      fprintf(stderr, "out of memory\n");
      exit(0);
   }

   q->items = malloc(size * sizeof(int)); // make room for the array
   if (q->items == NULL) {
      fprintf(stderr, "out of memory\n");
      exit(0);
   }
   q->nItems = 0;                               // we have no items yet
   q->maxsize = size;                           // remember the maxsize
   return q;                                    // return the initial PQ
}

void insertPQ(PQ q, int it) {
    if (q == NULL) {
       fprintf(stderr, "priority queue not initialised\n");
       exit(1);
    }
    if (q->nItems == q->maxsize) {
       fprintf(stderr, "priority queue full\n");
       exit(1);
    }
    q->items[q->nItems] = it; // UNORDERED ARRAY, so put item at the end
    q->nItems++;              // increment the 'counter'
}

int delMaxPQ(PQ q) { // UNORDERED, so need to linear search for max item
    if (q == NULL) {
       fprintf(stderr, "delmaxPQ: priority queue not initialised\n");
       exit(1);
    }
    if (q->nItems == 0) {
       fprintf(stderr, "priority queue empty\n");
       exit(1);
    }
    int *array = q->items;
    int last = q->nItems-1;       // items occupy places 0 .. last
    int max = 0;                  // assume initially item at max=0 has largest key
    for (int i = 1; i <= last; i++){
       if (array[max] < array[i]){ // now compare with every other item
          max = i;                 // whenever we find a better one, update max
       }
    }
```

```
    int retval = array[max];        // save the max item
    array[max] = array[last];       // overwrite max location with last item
    q->nItems--;                    // decrease the number of items
    return retval;                  // return the max element
}

int isEmptyPQ(PQ q) {
   int empty = 0;
   if (q == NULL) {
      fprintf(stderr, "isEmptyPQ: priority queue not initialised\n");
   }
   else {
      empty = q->nItems == 0;
   }
   return empty;
}
```

Compile and run the client *pqSort()* with this ADT:

```
prompt$ dcc pqUA.c pqSort.c
prompt$ ./a.out
Array: 41 2 58 156 360 81 260 74 167 13
Sorted: 360 260 167 156 81 74 58 41 13 2
```

Notice the descending order, produced by calling *delMaxPQ()* in the client.

PQ ADT implementation using an ordered array

We could have implemented the priority queue using an ADT based on an Ordered Array

- this will make it much easier to find the maximum element ...
- ... but when we insert, we will need to do more work

切换行号显示

```
// pqOA.c: priority queue implementation for pq.h using an ordered array
#include "pq.h"

struct pqRep {
   int nItems;  // actual count of Items
   int *items;  // array of Items
   int maxsize; // maximum size of array
};

PQ createPQ(int size) {
   PQ q = malloc(sizeof(struct pqRep));  // make room for the structure
   if (q == NULL) {
      fprintf(stderr, "out of memory\n");
      exit(0);
   }
   q->items = malloc(size * sizeof(int)); // make room for the array
   if (q->items == NULL) {
      fprintf(stderr, "out of memory\n");
      exit(0);
   }
   q->nItems = 0;                          // we have no items yet
   q->maxsize = size;                      // remember the maxsize
   return q;                               // return the initial PQ
}
```

```

void insertPQ(PQ q, int it) {
    if (q == NULL) {
       fprintf(stderr, "priority queue not initialised\n");
       exit(1);
    }
    if (q->nItems == q->maxsize) {
       fprintf(stderr, "priority queue full\n");
       exit(1);
    }
    int *array = q->items;
    int last = q->nItems;
    int i;
    for (i=0; i<last && array[i]<it; i++) {
        ;                           // find location of item == it
    }
    int j;
    for (j = last; j>i; j--){   // starting at last and go down to i
        array[j] = array[j-1]; // shift items up
    }
    array[i] = it;              // now insert item 'it' at i
    q->nItems++;                // increase the count
}

int delMaxPQ(PQ q) {
    if (q == NULL) {
       fprintf(stderr, "priority queue not initialised\n");
       exit(1);
    }
    if (q->nItems == 0) {
       fprintf(stderr, "priority queue empty\n");
       exit(1);
    }
    q->nItems--;
    return q->items[q->nItems];
}

int isEmptyPQ(PQ q) {
   int empty = 0;
   if (q == NULL) {
      fprintf(stderr, "isEmptyPQ: priority queue not initialised\n");
   }
   else {
      empty = q->nItems == 0;
   }
   return empty;
}
```

Compile and run the client *pqSort.c* with this ADT:

```
prompt$ dcc pqOA.c pqSort.c
prompt$ ./a.out
Array: 41 2 58 156 360 81 260 74 167 13
Sorted: 360 260 167 156 81 74 58 41 13 2
```

- the output is the same as for the unordered array

Example of use of the unordered and ordered array *PQ* implementations:

Operation	Ordered	Unordered	returned value
insertPQ E	E	E	-
insertPQ X	EX	EX	-

insertPQ A	AEX	EXA	-
insertPQ M	AEMX	EXAM	-
delMaxPQ	AEM	EMA	X

Note:

- you cannot see from 'outside the ADT' which order the items are in
- all you can see is the results of the *delMaxPQ* operation that removes them from the *PQ*
    - above, the same item 'X' is returned in both cases

Performance: ordered vs un-ordered

The performance of the 2 priority queue ADT implementations can be compared:

Implementation	insertPQ	delMaxPQ
ordered array	*O(N)*	*O(1)*
unordered array	*O(1)*	*O(N)*

Hence, we have poor performance either with an *insertPQ* or a *delMaxPQ*

- we know that, using a heap, we can *insert* and *delMax* in *O(log(n))*
- ***can we use a heap to implement a priority queue ADT?***

PQ implementation using a heap

A heap is also an array, like ordered and unordered

- (actually it could be some other data structure, but in this course it will always be an array)
- it contains the same elements, but in a different order than ordered and unordered
- the difference is the effect of *HOP* and *CTP*

切换行号显示

```
// pqHP.c: priority queue implementation for pq.h using a heap
#include "pq.h"

// 'static' means these functions are for local use only
static void fixDown(int *, int, int);
static void fixUp(int *, int);

// Priority queue implementation using an unordered array
struct pqRep {
   int nItems;  // actual count of Items
   int *items;  // array of Items
   int maxsize; // maximum size of array
};

PQ createPQ(int size) {
   PQ q = malloc(sizeof(struct pqRep));  // make room for the structure
   if (q == NULL) {
      fprintf(stderr, "out of memory\n");
      exit(0);
   }
   q->items = malloc((size+1) * sizeof(int)); // make room for the array
   if (q->items == NULL) {                    // size+1 because heap
```

```
1..size
        fprintf(stderr, "out of memory\n");
        exit(0);
    }
    q->nItems = 0;                          // we have no items yet
    q->maxsize = size;                      // remember the maxsize
    return q;                               // return the initial PQ
}

void insertPQ(PQ q, int it) {
    if (q == NULL) {
        fprintf(stderr, "priority queue not initialised\n");
        exit(1);
    }
    if (q->nItems == q->maxsize) {
        fprintf(stderr, "priority queue full\n");
        exit(1);
    }
    q->nItems++;                        // adding another item
    q->items[q->nItems] = it;           // put the item at the end
    fixUp(q->items, q->nItems);         // fixUp all the way to the root
    return;
}

int delMaxPQ(PQ q) {
    if (q == NULL) {
        fprintf(stderr, "priority queue not initialised\n");
        exit(1);
    }
    if (q->nItems == 0) {
        fprintf(stderr, "priority queue empty\n");
        exit(1);
    }
    int retval = q->items[1];           // this is the item we want to return
    q->items[1] = q->items[q->nItems]; // overwrite root by last item
    q->nItems--;                        // we are decreasing heap size by 1
    fixDown(q->items, 1, q->nItems);    // fixDown the new root
    return retval;
}

int isEmptyPQ(PQ q) {
    int empty = 0;
    if (q == NULL) {
        fprintf(stderr, "isEmptyPQ: priority queue not initialised\n");
    }
    else {
        empty = q->nItems == 0;
    }
    return empty;
}

// fix up the heap for the 'new' element child
void fixUp(int *heap, int child) {
    while (child>1 && heap[child/2]<heap[child]) {
        int swap = heap[child];         // if parent < child, do a swap
        heap[child] = heap[child/2];
        heap[child/2] = swap;
        child = child/2;                // become the parent
    }
    return;
}

```

```
// force value at a[par] into correct position
void fixDown(int *heap, int par, int len) {
   int finished = 0;
   while (2*par<=len && !finished) {// as long as you are within bounds
      int child = 2*par;          // the first child is here
      if (child<len && heap[child]<heap[child+1]) {
         child++;                 // choose larger of two children
      }
      if (heap[par]<heap[child]) { // if node is smaller than this child ...
         int swap = heap[child];   // if parent < child, do a swap
         heap[child] = heap[child/2];
         heap[child/2] = swap;
         par = child;             // ... and become this child
      }
      else {
         finished = 1;            // else we do not have to go any further
      }
   }
   return;
}
```

Compile and run:

```
prompt$ dcc pqHP.c pqSort.c
prompt$ ./a.out
Array: 41 2 58 156 360 81 260 74 167 13
Sorted: 360 260 167 156 81 74 58 41 13 2
```

- this is the same output as for the unordered and ordered arrays

Performance: unordered vs ordered vs heap-based

Cost of operations in a heap:

- tree height is *log(n)*
- each insert/delete requires at most *log(n)* compares/swaps on a path from root to leaf
- complexity is *O(log(n))* for *insertPQ()* and *delMaxPQ()*

For just one operation:

Implementation	insertPQ	delMaxPQ
ordered array	*O(n)*	*O(1)*
unordered array	*O(1)*	*O(n)*
heap array	*O(log(n))*	*O(log(n))*

Multiply by *n* for an array of length *n*.

Heap Sort

The client* pqSort() *using the heap array implementation is heap sort

- ... although it can be implemented more efficiently

The client:

- stores the unsorted list as a heap (ADT)
- repeatedly deletes the maximum element (ADT)
- heapifying each time (ADT)

- Heap sort is **not adaptive** (*this is bad*)
  - means that it is not faster if the data is already partially ordered
    - even if data is ordered, *fixUp* and *fixDown* may still need to go from root to leaf
- Heap sort is **not stable** (*this is bad*)
  - means that terms with the same value may change order
- Heap sort is **in-place** ... (this is good)
  - ... but *pqSort()* is not

Heap sort is*O(nlog(n))* (best, average and worst case behaviour)

- merge sort, quick sort as well, but quick sort has worst case $O(n^2)$ (rarely happens)
- Heap sort is <u>not</u> recursive (*this is good*), merge sort and quick sort <u>are</u> recursive
- heapsort is faster than mergesort (by a constant factor) and more space efficient
  - ... although in a distributed environment (lots of machines) merge sort is better
- quick sort is generally faster than heap sort (by a constant factor) for most data sets

If speed is critical, and adaptivity & stability do not matter, heap sort is usually preferred.

Heap sort animation

PQs (2019-07-11 16:59:40由AlbertNymeyer编辑)