Lab 12: Function Pointers & Algorithm Complexity — Solution

Aim
This lab class gives you an opportunity to:

- complete your eVALUate survey(s) for the unit;
- write programs with function pointers;
- explore priority queues; and
- practise determining the complexity of algorithms.

Part 2 — Function Pointers

Context
Priority queues are queues in which items are stored in an order governed by their 'priority'. The highest priority items are at the front of the queue and the lowest priority items are at the back, i.e. items are stored in descending order of priority. It is the `add()` function that facilitates this; all other functions are unchanged from a queue. In order to know how to compare two 'things' and determine which is the most important, `add()` must be passed a function which does the comparison. Hence `add()`'s function header is:

```
void add(prique p,void *o,bool(*greaterThan)(void *,void *));
```

Here `p` is the priority queue that the value `o` will be added to in an order determined by `greaterThan()` — which is a function accepting two 'things' and which returns `true` if the first is more important than the second and `false` otherwise.

The data structure being used to implement the ADT is a single-ended doubly-linked list. `prique` is a pointer to a struct containing a reference to the first node in the linked list. `dnode` is a pointer to a struct which contains a `data` field, a `next` field, and a `prev` field which refers to the node prior to the current one.

Tasks

1. Download the compressed project folder `Lab12.zip` from *MyLO* and after extracting all the files, open this project folder and open the project file (`Lab12.sln`).
2. Complete the implementation of the ADT by writing the `add()` function for the `prique.c` file.

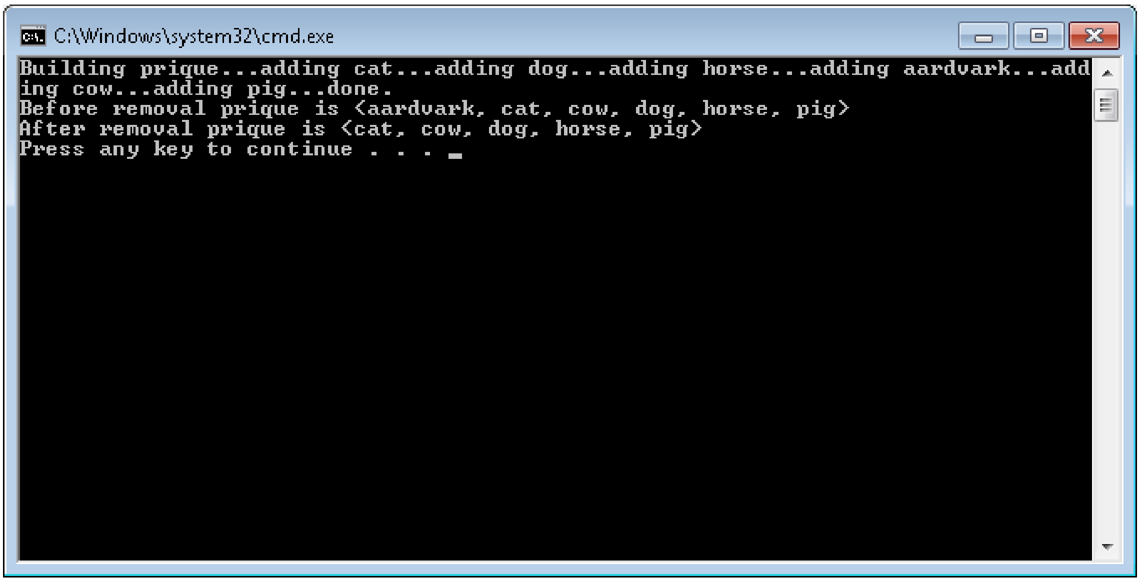
```
void add(prique p,void *o,bool(*greaterThan)(void *,void *))
{
    dnode n;      // new dnode
    dnode b,c;    // current and dnode prior to current

    init_dnode(&n,o);    // new dnode to add into prique
```

```
    if (isEmpty(p))       // prique was empty, now it's not
    {
        p->first=n;
    }
    else
    {
        c=p->first;       // traverse to determine location
        b=getPrev(c);
        while ((c!=NULL) && ((*greaterThan)(getData(c),o)))
        {
            b=c;
            c=getNext(c);
        }

        if (b==NULL)      // position is at head of prique
        {
            p->first=n;
        }
        else    // position is between dnodes or at the end
        {
            setNext(b,n);    // link previous on to the new one
        }
        setPrev(n,b);    // link new one back to the previous

        if (c!=NULL)      // not the last node
        {
            setPrev(c,n);    // link current back to new one
        }
        setNext(n,c);    // link new one on to the current
    }
}
```

3. Complete the driver application (`Lab12.c`) by writing the body of the `alphabeticalOrder()` function. *Hint*: `alphabeticalOrder()` can be implemented by calling `strcmp()`.

```
bool alphabeticalOrder(void *t1, void *t2)
{
    char *s1=(char *)t1;     // convert from void * to string
    char *s2=(char *)t2;

    return (strcmp(s1,s2) < 0);
}
```

4. Test that your implementation works. You should see the following output.



Part 3 — Algorithm Complexity

Context

Algorithms can be examined to determine the 'amount of work' they perform so that they may be compared with other algorithms completing the same task. The amount of work can vary from a minimum (best case) to a maximum (worst case). These complexities can then be expressed in order (big-oh) notation.

Task

1. Consider the following sorting algorithm implemented in C.

   Using "`(a[i] > a[i+1])`" as the fundamental operation, what are the best- and worst-case time complexities of the algorithm? What are these in big-oh notation? You might consider what happens when the list is already sorted and when the smallest item is at the rear of the list. Assume that the `swap()` function simply swaps the values at the given array locations.

```
void sort(int a[], int n)                         0
{                                                 0
    int i;                                        0
    bool exchange=true;                           0
                                                  0
    while (exchange) {                            ? x
        exchange=false;                               0
        for (i=0; i<n-1; i++) {                       ? x
            if (a[i] > a[i+1]) {                          1
                swap(a,i,i+1);                            0
                exchange=true;                            0
            }                                             0
        }                                                 0
    }                                                 0
}                                                 0
```

Given the annotations on the right to calculate the number of executions of the fundamental operation, the time complexity of `sort()` *is:*
T(n)=0+0+0+0+0+#while[0+#for*(1+0+0+0+0)+0]+0*
i.e. T(n)=#while#for*1.*

If the array is sorted no exchanges take place and the body of the `while` *loop is executed only once. Hence the body of the* `for` *loop is executed $n-1$ times. The best-case complexity is therefore $B(n)=n-1$ and is $O(n)$.*

If the last element in the array is the smallest it must be shuffled back to the beginning of the array. This can only happen one step at a time, thus the body of the `while` *loop is executed n times and the body of the* `for` *loop is executed $n-1$ times each time. Thus the worst-case complexity is $W(n)= n^2-n$ and is $O(n^2)$.*

Thus this function will never execute in less than $O(n)$ time or in greater than $O(n^2)$ time.