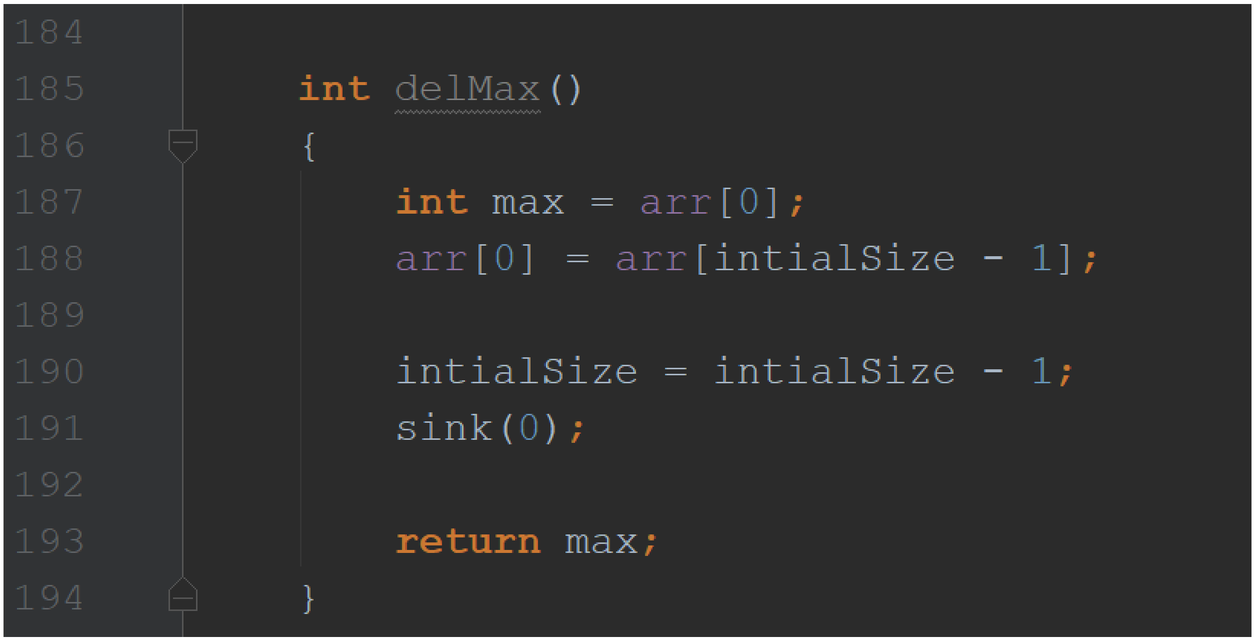
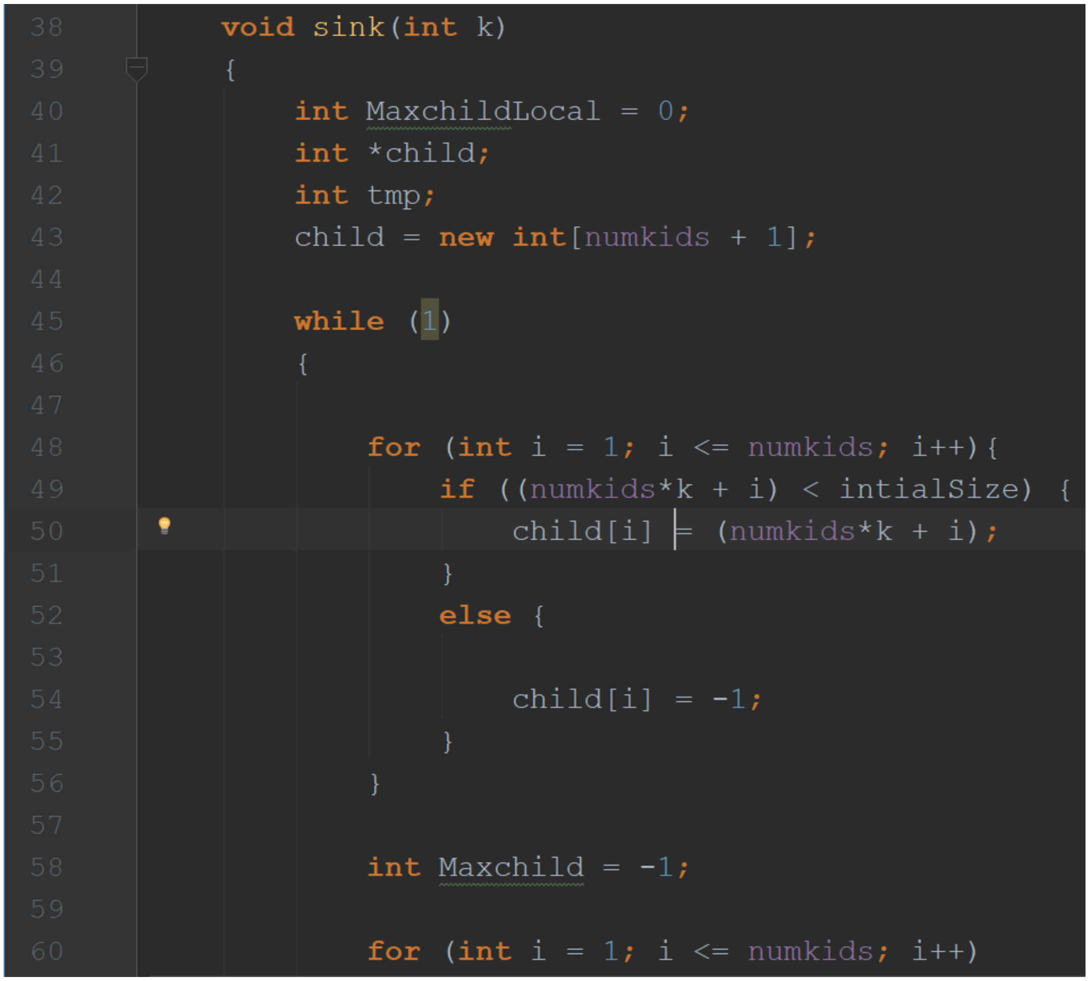
DelMax

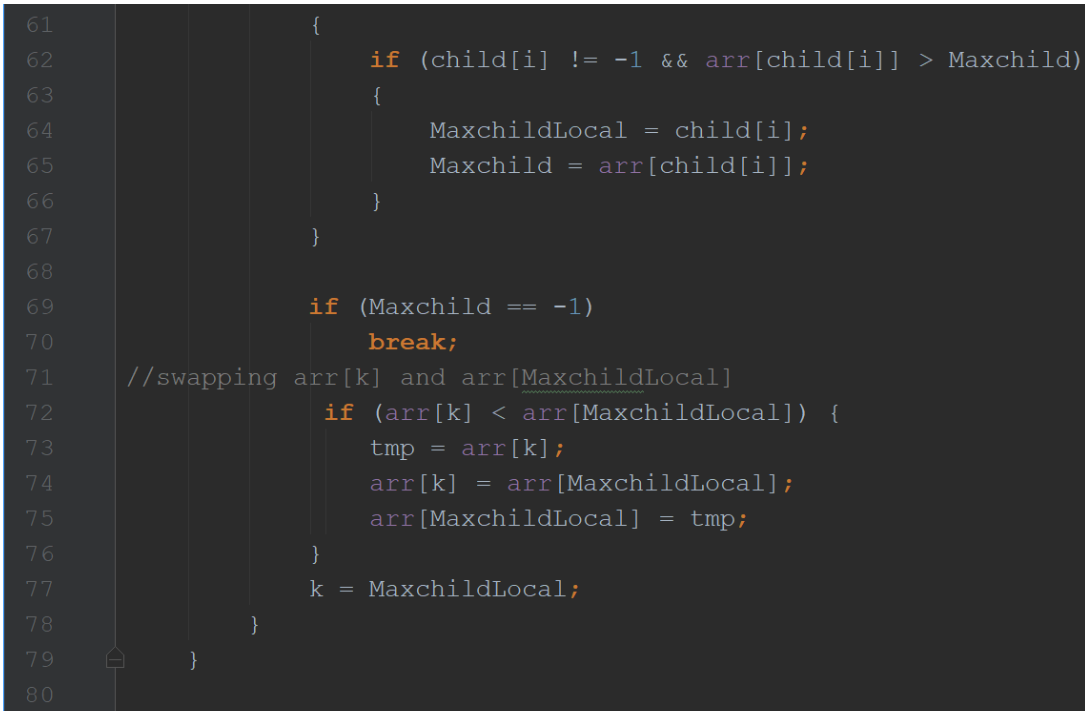


```
int delMax()
{
    int max = arr[0];
    arr[0] = arr[intialSize - 1];

    intialSize = intialSize - 1;
    sink(0);

    return max;
}
```

The function code for delMax does not contain any loops or recurring code and it only involves all constants therefore all the time complexity for delMax is a constant. However, the function delMax also includes the function sink. The screenshot of function sink is shown below.



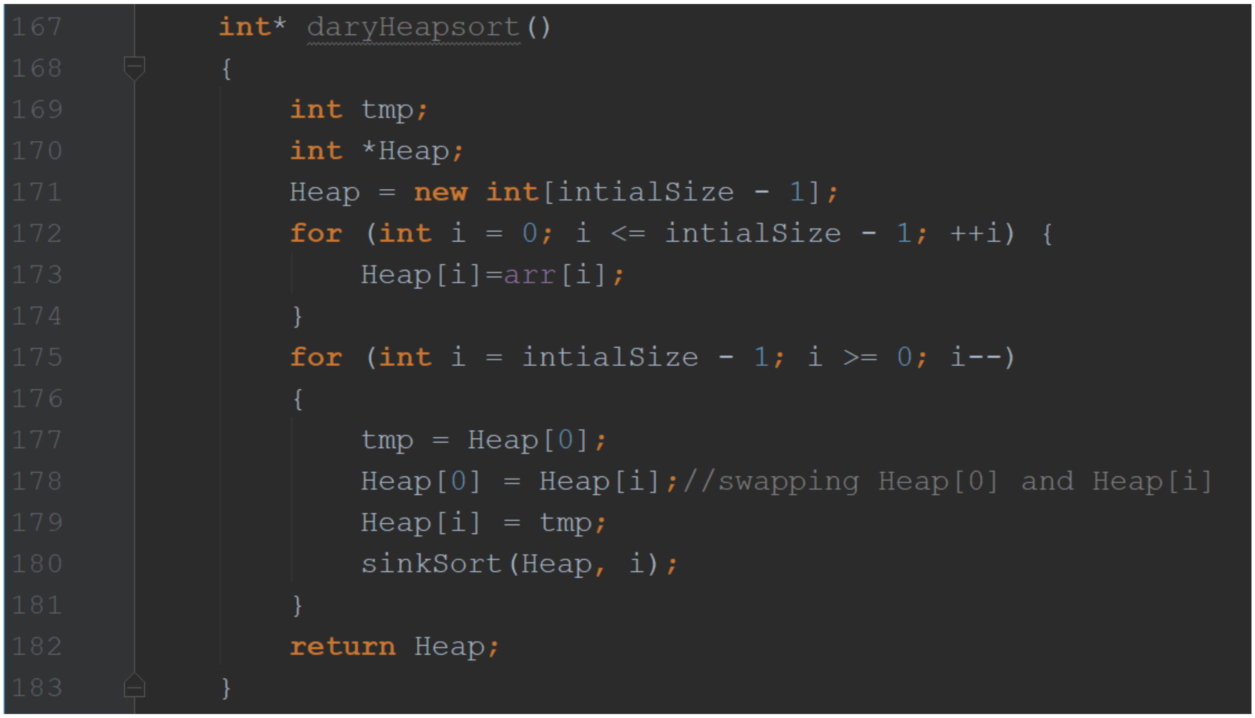
```
    void sink(int k)
    {
        int MaxchildLocal = 0;
        int *child;
        int tmp;
        child = new int[numkids + 1];

        while (1)
        {

            for (int i = 1; i <= numkids; i++){
                if ((numkids*k + i) < intialSize) {
                    child[i] = (numkids*k + i);
                }
                else {

                    child[i] = -1;
                }
            }

            int Maxchild = -1;

            for (int i = 1; i <= numkids; i++)
```



```
            {
                if (child[i] != -1 && arr[child[i]] > Maxchild)
                {
                    MaxchildLocal = child[i];
                    Maxchild = arr[child[i]];
                }
            }

            if (Maxchild == -1)
                break;
//swapping arr[k] and arr[MaxchildLocal]
             if (arr[k] < arr[MaxchildLocal]) {
                tmp = arr[k];
                arr[k] = arr[MaxchildLocal];
                arr[MaxchildLocal] = tmp;
            }
            k = MaxchildLocal;
        }
    }
```

In the code for sink function there are two for loop, and one while loop. But as we can see that the while loop will have to be called multiple times whenever the delMax function is called. The fuction of the while loop will run until the the node sinks to the end of the tree. So that will be the height of the tree, and the max height for us will be O(log_k(n)) where k is the number of children and n is the number of nodes in the heap. We also have for loops that will run when the while loop is ran so the for loops will run k times and we have 2 for loops so we will finally get O(log_k(n))*O(2k). But we can pull 2 out as constant. O(k*log_k(n)) would be the worst case. The while loop is at line 45 and for loops are at line 48 and line 60.

DaryHeapsort



```
int* daryHeapsort()
{
    int tmp;
    int *Heap;
    Heap = new int[intialSize - 1];
    for (int i = 0; i <= intialSize - 1; ++i) {
        Heap[i]=arr[i];
    }
    for (int i = intialSize - 1; i >= 0; i--)
    {
        tmp = Heap[0];
        Heap[0] = Heap[i];//swapping Heap[0] and Heap[i]
        Heap[i] = tmp;
        sinkSort(Heap, i);
    }
    return Heap;
}
```

As we can see there are two for loops and the worst time complexity for this will be a constant O(2n) however there is a sinkSort embedded in the code, so we will have to consider the time complexity of that as well. And the time complexity for sinkSort is O(k*log_k(n)) as shown in previous function. So we will get the total worst run time as O(k*log_k(n)).