```
Q1 void dualSearch (int A[], int size, int K, int dualIndex[])
    { int i=0,int j;
      for(i=0;i<size;i++)
          {for (j=i;j<size;j++)
              {if A[i]+A[j] == K
                  {return;}
              }
          }
    }

Time complexity  = n((n+1)/2) = n^2/2 + n/2 = θ(n^2)

n (n+n-1+..+1)

dualIndex[0] = i;
dualIndex[1] = j;
```

```
Q2 void dualSearch(int A[],int size, int K, int dualIndex[]){
    int j, int low = 0, int high = size-1;
    for(j=0;j<size;j++){
        diff = K - A[j];
        while low<=high:
            mid = (low+high)/2
            if A[mid] == diff
                return;
            else if diff>A[i]
                low = mid+1;
            else:
                high = mid-1;
```

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC1007 Data Structures and Algorithms **2022/23 Semester 2**

Tutorial 5: Searching and Hash Tables

School of Computer Science and Engineering *Nanyang Technological University*

Time complexity = n(c1+log2(n+c2))
=θ(nlog2n)

sequential search

Q1 Given an array of n elements. Find two elements in the array such that their sum is equal to K. The two elements can be the same element. Once a pair of elements is found, the program can be terminated. The function prototype is given below:

```
void dualSearch(int A[], int size, int K, int dualIndex[])
```

binary search

Q2 Given a sorted array of n elements. Find two elements in the array such that their sum is equal to K. The two elements can be the same element. Once a pair of elements are found, the program can be terminated. The results may be different from the results of Question 1. The function prototype is given below:

```
void dualSearch(int A[], int size, int K, int dualIndex[])
```

Q3 Compare the performance between Q1 and Q2. Try to use a large input file to evaluate their running time. If you are using Code::Blocks, you may use *Configure Tools* under *tools* to create a User-defined tools shown in Figure 5.1. You can also try to run your programs in the command line or terminal directly using I/O redirection "<" for standard input and ">" for standard output. A 500k data and 1 million data file are attached. The first line is a search key and the second line is a data size. The rest are the data.

Q4 Implement a closed addressing hash table to perform insertion and key searching. The insertion may not have to insert at the end of the link-list. The function prototype is given below:

```
ListNode* HashSearch(HashTable *, int);
int HashInsert(HashTable *, int);
```

```
void dualSearch(int A[],int size, int K, int dualIndex[]){
  int i,j,temp;

  i=0;
  j = size-1;
  while(i<j){
      temp = A[i]+A[j];

    if(temp==K){
      dualIndex[0] = i;
      dualIndex[1] = j;
      return;
    }
    else if(temp<K){
      i++;
    }
    else{
      j--;
    }}
```

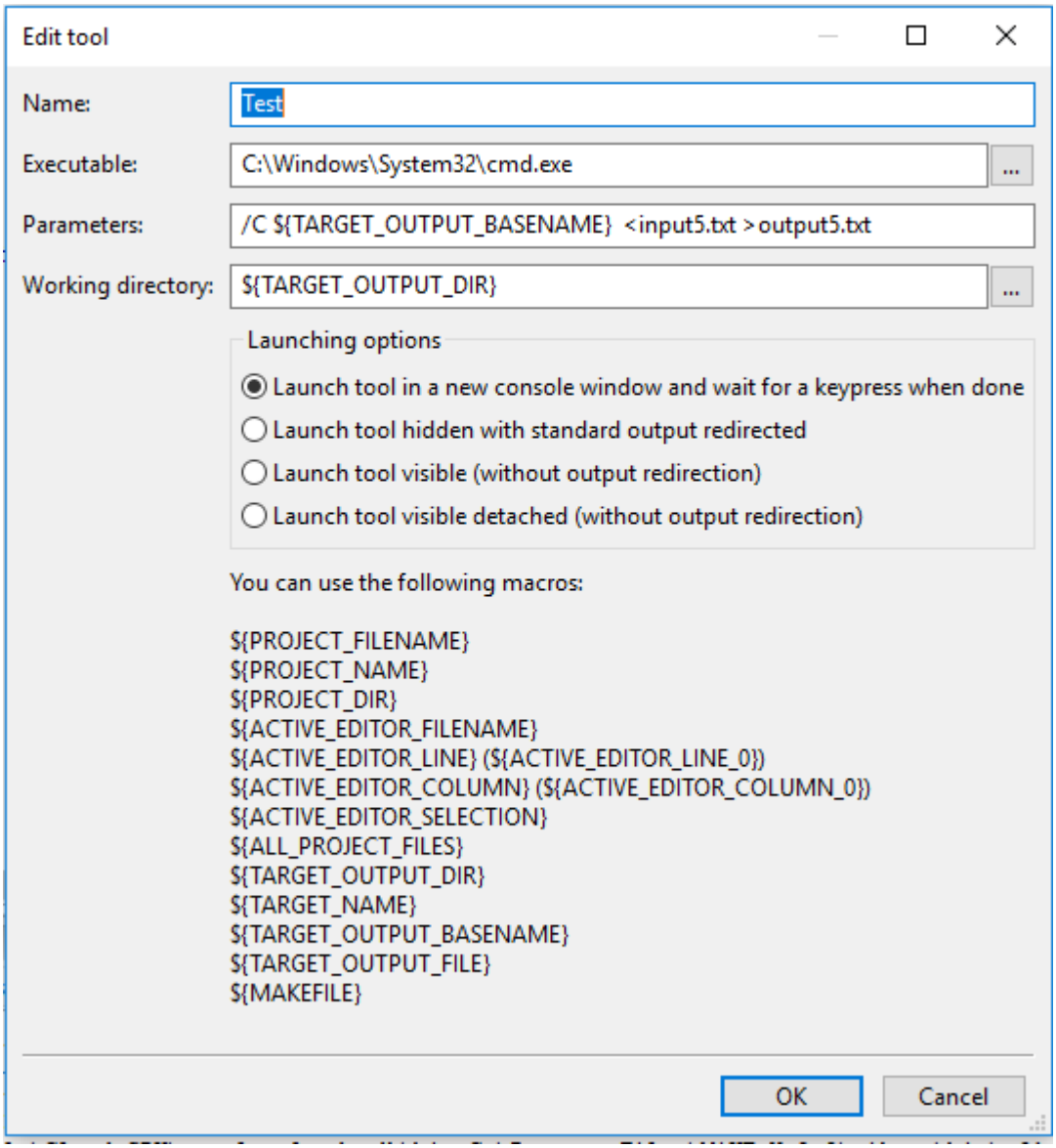

Figure 5.1: Name is not important. It can be anything. input5.txt is an input filename and output5.txt is an output filename.

```
ListNode* HashSearch(HashTable *,int){
int index;
                          value % to get index
                          go through linked list
ListNode *temp;

if(Q3Hash.hSize!=0)
   index = Hash(key,Q3Hash.hSize);
else
   return NULL;

temp = Q3Hash.Table[index].head;
while(temp!=NULL){
    if(temp->key==key)
        return temp;
    temp = temp->next;
}
return NULL;
}
```

```
int HashInsert(HashTable *,int){
int index;                                  value % to get the index
ListNode *newNode;                          insert in linked list format

if(HashSearch(*Q3HashPtr, key)!=NULL)
   return 0;

if(Q3HashPtr->hSize!=0)
  index = Hash(key,Q3HashPtr->hSize);

newNode = (ListNode *)malloc(sizeof(ListNode));
newNode->key = key;
newNode->next = Q3HashPtr->Table[index].head;
Q3HashPtr->Table[index].head = newNode;

Q3HashPtr->Table[index].size++;
Q3HashPtr->nSize++;

return 1;}
```