CTDL & GT

***Tìm kiếm nhị phân**

```
int binarySearch(int A[], int key, int left, int right)
{
	if (left > right)
		return -1;
	else{
		int mid = (left + right) / 2;

		if (A[mid] == key)
			return mid;

		if (A[mid] > key)
			return binarySearch(A, key, left, mid - 1);

		if (A[mid] < key)
			return binarySearch(A, key, mid + 1, right);
	}
}
```

==========

***Tìm kiếm nội suy**

```
// If x is present in arr[0..n-1], then returns index of it, else returns(-1)
int interpolationSearch(int arr[], int n, int x)
{
	// Find indexes of two corners
	int lo = 0, hi = (n - 1);

	// Since array is sorted, an element present
	// in array must be in range defined by corner
	while (lo <= hi && x >= arr[lo] && x <= arr[hi])
	{
		// Probing the position with keeping
		// uniform distribution in mind.
		int pos = lo + (((double)(hi - lo) /
			(arr[hi] - arr[lo]))*(x - arr[lo]));

		// Condition of target found
		if (arr[pos] == x)
			return pos;

		// If x is larger, x is in upper part
		if (arr[pos] < x)
			lo = pos + 1;

		// If x is smaller, x is in lower part
		else
			hi = pos - 1;
	}
	return -1;
}
```

***Bubble sort**

```
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

==========

***Selection sort**

```
void selectionSort(int a[], int n){
      int min;
      int temp;
      for (int i = 0; i < n; i++){
            min = i;
            for (int j = i + 1; j < n; j++){
                  if (a[j] < a[min])
                        min = j;
            }
            if (a[min] < a[i]) swap(a[min], a[i]);
      }
}
```

==========

***Insertion sort**

```
void insertionSort(int a[], int n){
      for (int i = 1; i < n; i++)
      {
            int x = a[i];
            int j = i;
            while (j > 0 && a[j - 1] > x)
            {
                  a[j] = a[j - 1];
                  j--;
            }
            a[j] = x;
      }
}
```

==========

***Shaker sort**

```
void ShakerSort(int a[], int n)
{
	int Left = 0;
	int Right = n - 1;
	int k = 0;
	while (Left < Right)
	{
		for (int i = Left; i < Right; i++)
		{
			if (a[i] > a[i + 1])
			{
				swap(a[i], a[i + 1]);
				k = i;
			}
		}
		Right = k;
		for (i = Right; i > Left; i--)
		{
			if (a[i] < a[i - 1])
			{
				swap(a[i], a[i - 1]);
				k = i;
			}
		}
		Left = k;
	}
}
```

==========

***Shell sort**

```
int shellSort(int arr[], int n)
{
	// Start with a big gap, then reduce the gap
	for (int gap = n / 2; gap > 0; gap /= 2)
	{
		// Do a gapped insertion sort for this gap size.
		// The first gap elements a[0..gap-1] are already in gapped order
		// keep adding one more element until the entire array is
		// gap sorted
		for (int i = gap; i < n; i += 1)
		{
			// add a[i] to the elements that have been gap sorted
			// save a[i] in temp and make a hole at position i
			int temp = arr[i];

			// shift earlier gap-sorted elements up until correct
			// location for a[i] is found
			int j;
			for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
```

```
                        arr[j] = arr[j - gap];

                  //  put temp (the original a[i]) in its correct location
                  arr[j] = temp;
            }
      }
      return 0;
}
```

==========

***Heap sort**

```
void shift(int a[], int left, int right)
{
      int i = left;
      int j = 2 * i;
      int x = a[i];

      while (j <= right)
      {
            if (j < right)
            {
                  if (a[j] > a[j + 1]) j++;
            }
            if (x <= a[j]) break;

            a[i] = a[j];
            i = j;
            j = i * 2;
      }
      a[i] = x;
}
void buildHeap(int a[], int n)
{
      int left = n / 2 - 1;
      while (left >= 0){
            shift(a, left, n);
            left--;
      }
}
void heapSort(int a[], int n)
{
      buildHeap(a, n);
      int right = n - 1;
      while (right > 0){
            swap(a[0], a[right]);
            right--;
            shift(a, 0, right);
      }
}
```

==========

***Quick sort**

```
int partition(int arr[], int left, int right)
{
	int i = left, j = right;
	int tmp;
	int pivot = arr[(left + right) / 2];

	while (i <= j) {
		while (arr[i] < pivot)
			i++;
		while (arr[j] > pivot)
			j--;
		if (i <= j) {
			tmp = arr[i];
			arr[i] = arr[j];
			arr[j] = tmp;
			i++;
			j--;
		}
	};
	return i;
}

void quickSort(int arr[], int left, int right) {
	int index = partition(arr, left, right);
	if (left < index - 1)
		quickSort(arr, left, index - 1);
	if (index < right)
		quickSort(arr, index, right);
}
```

==========

***Merge sort**

```
void Merge(int *a, int low, int high, int mid)
{
	// We have low to mid and mid+1 to high already sorted.
	int i, j, k, *temp;
	temp = new int[high - low + 1];
	i = low;
	k = 0;
	j = mid + 1;

	// Merge the two parts into temp[].
	while (i <= mid && j <= high)
	{
		if (a[i] < a[j])
		{
			temp[k] = a[i];
			k++;
			i++;
		}
		else
		{
			temp[k] = a[j];
			k++;
			j++;
		}
	}

	// Insert all the remaining values from i to mid into temp[].
	while (i <= mid)
	{
		temp[k] = a[i];
		k++;
		i++;
	}

	// Insert all the remaining values from j to high into temp[].
	while (j <= high)
	{
		temp[k] = a[j];
		k++;
		j++;
	}


	// Assign sorted data stored in temp[] to a[].
	for (i = low; i <= high; i++)
	{
		a[i] = temp[i - low];
	}
}
```

```
// A function to split array into two parts.
void MergeSort(int *a, int low, int high)
{
	int mid;
	if (low < high)
	{
		mid = (low + high) / 2;
		// Split the data into two half.
		MergeSort(a, low, mid);
		MergeSort(a, mid + 1, high);

		// Merge them to get sorted output.
		Merge(a, low, high, mid);
	}
}
```

```
void MergeSort(int *a, int low, int high)
{
	int mid;
	if (high - low + 1 <= 5)
	{
		sortfive(a, low, high - low + 1);
		return;
	}
	if (low < high)
	{
		mid = (low + high) / 2;
		if (mid - low + 1 <= 5)
			sortfive(a, low, mid - low + 1);
		if (high - mid + 1 <= 5)
			sortfive(a, mid, high - mid + 1);
		// Split the data into two half.

		MergeSort(a, low, mid);
		MergeSort(a, mid + 1, high);

		// Merge them to get sorted output.

		Merge(a, low, high, mid);
	}
}
```

==========

***Counting sort**

```
void countSort(char arr[])
{
    // The output character array that will have sorted arr
    char output[strlen(arr)];

    // Create a count array to store count of inidividual
    // characters and initialize count array as 0
    int count[RANGE + 1], i;
    memset(count, 0, sizeof(count));

    // Store count of each character
    for(i = 0; arr[i]; ++i)
        ++count[arr[i]];

    // Change count[i] so that count[i] now contains actual
    // position of this character in output array
    for (i = 1; i <= RANGE; ++i)
        count[i] += count[i-1];

    // Build the output character array
    for (i = 0; arr[i]; ++i)
    {
        output[count[arr[i]]-1] = arr[i];
        --count[arr[i]];
    }

    // Copy the output array to arr, so that arr now
    // contains sorted characters
    for (i = 0; arr[i]; ++i)
        arr[i] = output[i];
}
```

****

```
void countSort(int a[], int n)
{
      int CountArr[10] = { 0 };

      for (int i = 0; i < n; i++)
      {
            CountArr[a[i]]++;
      }

      int outputindex = 0;
      for (int j = 0; j < 10; j++)
      {
            while (CountArr[j]--)
                  a[outputindex++] = j;
      }
}
```

==========

***Radix sort**

~C1: Có cấp phát 10 ô nhớ

```
int getMax(int *a, int n)
{
	int max = a[0];
	for (int i = 1; i < n; i++)
		if (a[i] > max)
			max = a[i];
	return max;
}

//base: co so
int digit(int x, int k)
{
	int exp = pow(10, k - 1);
	return (x / exp) % 10;
}
void rSort(int *a, int n, int k)
{
	int **L = new int*[10];

	int *len = new int[10]{ 0 };
	for (int i = 0; i < n; i++)
	{
		len[digit(a[i], k)]++;
	}

	for (int i = 0; i < 10; i++)
	{
		if (len[i] > 0)
		{
			L[i] = new int[len[i]]{ 0 };
		}
		else L[i] = NULL;
	}
	delete[] len;

	int *t = new int[10]{ 0 };
	for (int i = 0; i < n; i++)
	{
		int j = digit(a[i], k);
		L[j][t[j]++] = a[i];
	}
```

```
	int num = 0;
	for (int i = 0; i < 10; i++)
	{
		for (int j = 0; j < t[i]; j++)
		{
			a[num++] = L[i][j];
		}
	}
	delete[] t;

	for (int i = 0; i < 10; i++)
	{
		delete[] L[i];
	}
	delete[] L;
}

//radix sort tu phai sang trai
void radixSort(int *a, int n)
{
	int max = getMax(a, n);
	int d = 0;
	while (max > 0)
	{
		max /= 10;
		d++;
	}

	for (int i = 1; i <= d; i++)
	{
		rSort(a, n, i);
	}
}
```

~Radix sort RECURSION

```
void rSort(int *a, int *b, int l, int r, int k)
{
	if (k < 0)return;

	int *f = new int[11]{ 0 };

	for (int i = l; i <= r; i++)
	{
		f[digit(a[i], k) + 1]++;
	}

	for (int i = 1; i < 11; i++)
	{
		f[i] += f[i - 1];
	}

	for (int i = l; i <= r; i++)
	{
		int j = digit(a[i], k);
		b[f[j]++] = a[i];
	}

	for (int i = l; i <= r; i++)
	{
		a[i] = b[i - l];
	}

	rSort(a, b, l, l + f[0] - 1, k - 1);

	for (int i = 0; i < 10; i++)
	{
		rSort(a, b, l + f[i], l + f[i + 1] - 1, k - 1);
	}
	delete[] f;
}

void radixSort(int *a, int n)
{
	int max = getMax(a, n);
	int d = (int)log((double)max) + 1;

	int *b = new int[n];
	rSort(a, b, 0, n - 1, d - 1);
	delete[] b;
}
```

==========

***Flash sort**

```
void flashSort(int *a, int n){
	if (n == 0)
		return;
	int m = (int)((0.2*n) + 2);

	int min, max, maxIndex;
	min = max = a[0];
	maxIndex = 0;

	for (int i = 1; i < n; i += 2){
		int small;
		int big;
		int bigIndex;

		if (a[i] < a[i + 1]){
			small = a[i];
			big = a[i + 1];
			bigIndex = i + 1;
		}
		else
		{
			big = a[i];
			bigIndex = i + 1;
			small = a[i + 1];
		}

		if (big > max){
			max = big;
			maxIndex = bigIndex;
		}

		if (small < min){
			min = small;
		}
	}

	if (a[n - 1] < min){
		min = a[n - 1];
	}
	else if (a[n - 1]>max){
		max = a[n - 1];
		maxIndex = n - 1;
	}

	if (max == min){
		return;
	}

	int* L = new int[m + 1];
```

```
for (int t = 0; t <= m; t++){
      L[t] = 0;
}

double c = (m - 1.0) / (max - min);
int K;
for (int h = 0; h < n; h++){
      K = ((int)((a[h] - min)*c)) + 1;
      L[K] += 1;
}

for (K = 2; K <= m; K++){
      L[K] = L[K] + L[K - 1];
}

int temp = a[maxIndex];
a[maxIndex] = a[0];
a[0] = temp;

int j = 0;
K = m;

int numMoves = 0;
while (numMoves < n){
      while (j >= L[K]){
            j++;
            K = ((int)((a[j] - min)*c)) + 1;
      }

      int evicted = a[j];
      while (j < L[K]){

            K = ((int)((evicted - min)*c)) + 1;
            int location = L[K] - 1;


            int temp = a[location];
            a[location] = evicted;
            evicted = temp;


            L[K] -= 1;
            numMoves++;
      }
}
```

```
      int threshold = (int)(1.25*((n / m) + 1));

      const int minElements = 30;
      for (K = m - 1; K >= 1; K--){
            int classSize = L[K + 1] - L[K];
            if (classSize > threshold && classSize > minElements){
                  flashSort(&a[L[K]], classSize);
            }
            else
            {
                  if (classSize > 1){
                        insertionSort(&a[L[K]], classSize);
                  }
            }
      }
      delete[]L;
}
```

****Strassen**

```
#include <iostream>

using namespace std;

const int N = 6;     //Define the size of the Matrix

template<typename T>
void Strassen(int n, T A[][N], T B[][N], T C[][N]);

template<typename T>
void input(int n, T p[][N]);

template<typename T>
void output(int n, T C[][N]);

int main() {
    //Define three Matrices
    int A[N][N],B[N][N],C[N][N];

    //?A?B????,???????,???
    for(int i=0; i<N; i++) {
       for(int j=0; j<N; j++) {
          A[i][j] = i * j;
          B[i][j] = i * j;
       }
    }

    //??Strassen????C=A*B
    Strassen(N, A, B, C);

    //????C??
```

```
    output(N, C);

    system("pause");
    return 0;
}

/**The Input Function of Matrix*/
template<typename T>
void input(int n, T p[][N]) {
     for(int i=0; i<n; i++) {
        cout<<"Please Input Line "<<i+1<<endl;
        for(int j=0; j<n; j++) {
           cin>>p[i][j];
        }
     }
}

/**The Output Fanction of Matrix*/
template<typename T>
void output(int n, T C[][N]) {
     cout<<"The Output Matrix is :"<<endl;
     for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
           cout<<C[i][j]<<" "<<endl;
        }
     }
}

/**Matrix Multiplication as the normal algorithm*/
template<typename T>
void Matrix_Multiply(T A[][N], T B[][N], T C[][N]) {  //Calculating A*B->C
     for(int i=0; i<2; i++) {
        for(int j=0; j<2; j++) {
           C[i][j] = 0;
           for(int t=0; t<2; t++) {
              C[i][j] = C[i][j] + A[i][t]*B[t][j];
           }
        }
     }
}

/**Matrix Addition*/
template <typename T>
void Matrix_Add(int n, T X[][N], T Y[][N], T Z[][N]) {
     for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
           Z[i][j] = X[i][j] + Y[i][j];
        }
     }
}
```

```
/**Matrix Subtraction*/
template <typename T>
void Matrix_Sub(int n, T X[][N], T Y[][N], T Z[][N]) {
     for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
           Z[i][j] = X[i][j] - Y[i][j];
        }
     }
}

/**
* ??n????A,B,C???,????????Strassen??
* ??A,B,C???????N??????????
*/
template <typename T>
void Strassen(int n, T A[][N], T B[][N], T C[][N]) {
     T A11[N][N], A12[N][N], A21[N][N], A22[N][N];
     T B11[N][N], B12[N][N], B21[N][N], B22[N][N];
     T C11[N][N], C12[N][N], C21[N][N], C22[N][N];
     T M1[N][N], M2[N][N], M3[N][N], M4[N][N], M5[N][N], M6[N][N],
M7[N][N];
     T AA[N][N], BB[N][N];

     if(n == 2) {  //2-order
        Matrix_Multiply(A, B, C);
     } else {
        //???A?B???????????,??????
        for(int i=0; i<n/2; i++) {
           for(int j=0; j<n/2; j++) {
              A11[i][j] = A[i][j];
              A12[i][j] = A[i][j+n/2];
              A21[i][j] = A[i+n/2][j];
              A22[i][j] = A[i+n/2][j+n/2];

              B11[i][j] = B[i][j];
              B12[i][j] = B[i][j+n/2];
              B21[i][j] = B[i+n/2][j];
              B22[i][j] = B[i+n/2][j+n/2];
           }
        }

        //Calculate M1 = (A0 + A3) × (B0 + B3)
        Matrix_Add(n/2, A11, A22, AA);
        Matrix_Add(n/2, B11, B22, BB);
        Strassen(n/2, AA, BB, M1);

        //Calculate M2 = (A2 + A3) × B0
        Matrix_Add(n/2, A21, A22, AA);
        Strassen(n/2, AA, B11, M2);
```

```
        //Calculate M3 = A0 × (B1 - B3)
        Matrix_Sub(n/2, B12, B22, BB);
        Strassen(n/2, A11, BB, M3);

        //Calculate M4 = A3 × (B2 - B0)
        Matrix_Sub(n/2, B21, B11, BB);
        Strassen(n/2, A22, BB, M4);

        //Calculate M5 = (A0 + A1) × B3
        Matrix_Add(n/2, A11, A12, AA);
        Strassen(n/2, AA, B22, M5);

        //Calculate M6 = (A2 - A0) × (B0 + B1)
        Matrix_Sub(n/2, A21, A11, AA);
        Matrix_Add(n/2, B11, B12, BB);
        Strassen(n/2, AA, BB, M6);

        //Calculate M7 = (A1 - A3) × (B2 + B3)
        Matrix_Sub(n/2, A12, A22, AA);
        Matrix_Add(n/2, B21, B22, BB);
        Strassen(n/2, AA, BB, M7);

        //Calculate C0 = M1 + M4 - M5 + M7
        Matrix_Add(n/2, M1, M4, AA);
        Matrix_Sub(n/2, M7, M5, BB);
        Matrix_Add(n/2, AA, BB, C11);

        //Calculate C1 = M3 + M5
        Matrix_Add(n/2, M3, M5, C12);

        //Calculate C2 = M2 + M4
        Matrix_Add(n/2, M2, M4, C21);

        //Calculate C3 = M1 - M2 + M3 + M6
        Matrix_Sub(n/2, M1, M2, AA);
        Matrix_Add(n/2, M3, M6, BB);
        Matrix_Add(n/2, AA, BB, C22);

        //Set the result to C[][N]
        for(int i=0; i<n/2; i++) {
           for(int j=0; j<n/2; j++) {
              C[i][j] = C11[i][j];
              C[i][j+n/2] = C12[i][j];
              C[i+n/2][j] = C21[i][j];
              C[i+n/2][j+n/2] = C22[i][j];
           }
        }
     }
}=====================================================================
```

***String search Brute Force**

```
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i+j] != pat[j])
                break;

        if (j == M)  // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
           printf("Pattern found at index %d n", i);
    }
}
```

***DFA String Search**

```
// C program for Finite Automata Pattern searching
// Algorithm
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)
{
    // If the character c is same as next character
    // in pattern,then simply increment state
    if (state < M && x == pat[state])
        return state+1;

    // ns stores the result which is next state
    int ns, i;

    // ns finally contains the longest prefix
    // which is also suffix in "pat[0..state-1]c"

    // Start from the largest possible value
    // and stop when you find a prefix which
    // is also suffix
    for (ns = state; ns > 0; ns--)
    {

        if (pat[ns-1] == x)
        {
```

```
            for (i = 0; i < ns-1; i++)
                if (pat[i] != pat[state-ns+1+i])
                    break;
            if (i == ns-1)
                return ns;
        }
    }
    return 0;
}

/* This function builds the TF table which represents
    Finite Automata for a given pattern */
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int state, x;
    for (state = 0; state <= M; ++state)
        for (x = 0; x < NO_OF_CHARS; ++x)
            TF[state][x] = getNextState(pat, M, state, x);
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTF(pat, M, TF);

    // Process txt over FA.
    int i, state=0;
    for (i = 0; i < N; i++)
    {
        state = TF[state][txt[i]];
        if (state == M)
            printf ("\n Pattern found at index %d", i-M+1);
    }
}

// Driver program to test above function
int main()
{
    char *txt = "AABAACAADAABAAABAA";
    char *pat = "AABA";
    search(pat, txt);
    return 0;
}
```

==========

***Horspool**

```
#define SIZE 256

void computeArray(char* pat, int* D)
{
	int M = strlen(pat);
	for (int c = 0; c < SIZE; c++)
	{
		D[c] = M;
		for (int i = 0; i < M - 1; i++) //ko xet ky tu cuoi cua pattern
		{
			D[pat[i]] = M - 1 - i;
		}
	}
}

void Horspool(char* pat, char* txt)
{
	int M = strlen(pat), N = strlen(txt);
	int* D = new int[SIZE];
	computeArray(pat, D);

	int i = M - 1;
	while (i < N)
	{
		int k = 0;
		while (k < M && pat[M - 1 - k] == txt[i - k]) k++;
		if (k == M) cout << "\nPos = " << i - M + 1;
		i += D[txt[i]];
	}
	delete D;
}
```

==========

*KMP

```
void computeArray(char* pat, int* lps)
{
	int M = strlen(pat);

	int k;
	lps[0] = lps[1] = k = 0;

	for (int q = 2; q <= M; q++)
	{
		while (k > 0 && pat[k] != pat[q - 1])
		{
			k = lps[k];
		}
		if (pat[k] == pat[q - 1])k++;
		lps[q] = k;
	}
}
```

```
void KMP(char* pat, char* txt)
{
        int M = strlen(pat);
        int N = strlen(txt);

        int* lps = new int[M + 1];

        computeArray(pat, lps);
        int q = 0;

        for (int i = 0; i < N; i++)
        {
                while (q > 0 && pat[q] != txt[i])
                {
                        q = lps[q];
                }
                if (pat[q] == txt[i])q++;
                if (q == M)
                {
                        cout << "\nPos = " << i - M + 1;
                        q = lps[q];
                }
        }
        delete lps;
}
```

==========

***Rabin Karp**

```
// d is the number of characters in input alphabet
#define d 256

/* pat -> pattern
txt -> text
q -> A prime number
*/
void search(char pat[], char txt[], int q)
{
      int M = strlen(pat);
      int N = strlen(txt);
      int i, j;
      int p = 0; // hash value for pattern
      int t = 0; // hash value for txt
      int h = 1;

      // The value of h would be "pow(d, M-1)%q"
      for (i = 0; i < M - 1; i++)
            h = (h*d) % q;
```

```
	// Calculate the hash value of pattern and first
	// window of text
	for (i = 0; i < M; i++)
	{
		p = (d*p + pat[i]) % q;
		t = (d*t + txt[i]) % q;
	}

	// Slide the pattern over text one by one
	for (i = 0; i <= N - M; i++)
	{

		// Check the hash values of current window of text
		// and pattern. If the hash values match then only
		// check for characters on by one
		if (p == t)
		{
			/* Check for characters one by one */
			for (j = 0; j < M; j++)
			{
				if (txt[i + j] != pat[j])
					break;
			}

			// if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
			if (j == M)
				printf("Pattern found at index %d \n", i);
		}

		// Calculate hash value for next window of text: Remove
		// leading digit, add trailing digit
		if (i < N - M)
		{
			t = (d*(t - txt[i] * h) + txt[i + M]) % q;

			// We might get negative value of t, converting it
			// to positive
			if (t < 0)
				t = (t + q);
		}
	}
}
```

==========

**Single linked list Basic

```
typedef struct Node * ref;
struct Node
{
	int key;
	Node* next;
};


Node* getNode(int k)
{
	Node* p = new Node;
	if (p == NULL)return NULL;
	p->key = k;
	p->next = NULL;
	return p;
}

//C1: tuan tu
void nhapMangTang(Node* &h, int &count)
{
	int n;
	cout << "\nNhap so luong: ";
	cin >> n;

	count = 0;
	h = getNode(-1);

	int k;
	for (int i = 0; i < n; i++)
	{
		cout << "\nNhap phan tu thu " << i + 1 << ": ";
		cin >> k;

		Node* p2 = h, *p1 = h->next;
		while (p1!=NULL && p1->key <= k)
		{
			if (p1->key == k)count++;
			p2 = p1;
			p1 = p1->next;
		}

		if (p2->key != k)
		{
			Node* p = getNode(k);
			p2->next = p;
			p->next = p1;
		}
	}
}
```

```
//C2: linh canh
void nhapMangTang2(Node* &h, int &count)
{
	int n;
	cout << "\nNhap so luong: ";
	cin >> n;

	count = 0;
	h = getNode(-1);

	int k;
	for (int i = 0; i < n; i++)
	{
		cout << "\nNhap phan tu thu " << i + 1 << ": ";
		cin >> k;

		Node* p2 = h, *p1 = h->next;
		while (p1 != NULL && p1->key < k)
		{
			p2 = p1;
			p1 = p1->next;
		}
		Node* p = getNode(k);
		p2->next = p;
		p->next = p1;
	}
}

void xuatMang(Node* h)
{
	for (Node* p = h; p != NULL; p = p->next)
	{
		cout << p->key << "   ";
	}
}
```

==========

****Single linked list – interchange sort**

```
struct Node
{
	int key;
	Node* next;
};

Node* getNode(int k)
{
	Node* p = new Node;
	if (p == NULL)return NULL;
	p->key = k;
	p->next = NULL;
	return p;
```

```
}

void push(Node* &head, int k)
{
	Node* p = getNode(k);
	if (head == NULL)
	{
		head = p;
	}
	else
	{
		p->next = head;
		head = p;
	}
}

Node* getTail(Node* head)
{
	Node* p = head;
	for (; p != NULL && p->next != NULL; p = p->next)
	{
	}
	return p;
}

void printList(Node* head)
{
	Node* p = head;
	while (p != NULL)
	{
		cout << p->key << "   ";
		p = p->next;
	}
	cout << endl;
}
void interchangeSort(Node *&head)
{
	if (head == NULL || head->next == NULL)return; //dslk co 0 hoac 1 phan
tu

	Node *curX, *prevX = NULL;
	Node *curY, *prevY = head; // curY=curX->next

	for (curX = head; curX->next != NULL; curX = curX->next)
	{
		prevY = curX;
		for (curY = curX->next; curY != NULL; curY = curY->next)
		{
			if (curX->key > curY->key)
			{
				if (prevX != NULL)
				{
					prevX->next = curY;
				}
				else
				{
```

```
                                        head = curY;
                                }

                                if (prevY != NULL)
                                {
                                        prevY->next = curX;
                                }
                                else
                                {
                                        head = curX;
                                }

                                Node *temp = curY->next;
                                curY->next = curX->next;
                                curX->next = temp;

                                //cap nhat
                                Node* t = curY;
                                curY = curX;
                                curX = t;

                        }
                        prevY = curY;
                }
                prevX = curX;
        }
}
```

==========

**Single linked list - Quick sort

```
void addFirst(Ref &h, Ref &t, int k)
{
      Ref p = getNode(k);

      if (h == NULL) {
            h = t = p;
            t->next = NULL;
      }
      else
      {
            p->next = h;
            h = p;
      }
}

void quickSort(Ref& h, Ref& t)
{
      if (h == t)
            return;

      Ref h1 = NULL, h2 = NULL, t1 = h1, t2 = h2;

      Ref tag = h;
```

```
	h = h->next;
	tag->next = NULL;

	while (h)
	{
		Ref p = h;
		h = h->next;
		p->next = NULL;
		if (p->data < tag->data)
			addFirst(h1, t1, p->data);
		else
			addFirst(h2, t2, p->data);
	}

	quickSort(h1, t1);
	quickSort(h2, t2);

	if (h1 != NULL) {
		h = h1;
		t1->next = tag;
	}
	else h = tag;
	tag->next = h2;
	if (h2 != NULL) {
		t = t2;
	}
	else t = tag;
}
```

==========

****Single linked list - Radix sort**

```
struct Node
{
	int key;
	Node* next;
};

Node* getNode(int k)
{
	Node* p = new Node;
	if (p == NULL)return NULL;
	p->key = k;
	p->next = NULL;
	return p;
}

void push(Node* &head, int k)
{
	Node* p = getNode(k);
	if (head == NULL)
	{
		head = p;
	}
```

```
        else
        {
                p->next = head;
                head = p;
        }
}

Node* getTail(Node* head)
{
        Node* p = head;
        for (; p!=NULL && p->next != NULL; p = p->next)
        {
        }
        return p;
}

void printList(Node* head)
{
        Node* p = head;
        while (p != NULL)
        {
                cout << p->key << "   ";
                p = p->next;
        }
        cout << endl;
}

void radixSort(Ref& h, Ref& t)
{
      int m = 10;

      if (h == t) return;

      int max=h->data, d;

      Ref q = h->next;
      while (q) {
            if (q->data > max)
                  max = q->data;
            q = q->next;
      }
      d = CountDigit(max);

      for (int i = 1; i <= d; i++)
      {
            Ref head[10], tail[10];
            for (int t = 0; t < 10; t++)
            {
                  head[t] = NULL;
                  tail[t] = head[t];
            }

            while (h)
```

```
            {
                  Ref p = h;
                  h = h->next;
                  p->next = NULL;
                  int k = digit(p->data,i);
                  addTail(head[k], tail[k], p);
            }

            for (int j = 0; j < 10; j++)
            {
                  if (h == NULL) {
                        if (head[j] != NULL) {
                              h = head[j];
                              t = tail[j];
                              t->next = NULL;
                        }
                  }
                  else
                  {
                        if (head[j] != NULL) {
                              t->next = head[j];
                              t = tail[j];
                              t->next = NULL;
                        }
                  }
            }
            cout << endl;
            printList(h);
      }

}
```

C2: Qsort

```
// Partitions the list taking the last element as the pivot
Node *partition(struct Node *head, struct Node *end, struct Node *&newHead,
struct Node *&newEnd)
{
        Node *pivot = end;
        Node *prev = NULL, *cur = head, *tail = pivot;

        // During partition, both the head and end of the list might change
        // which is updated in the newHead and newEnd variables
        while (cur != pivot)
        {
                if (cur->key < pivot->key)
                {
                        // First node that has a value less than the pivot -
becomes
                        // the new head
                        if ((newHead) == NULL)
                                (newHead) = cur;
```

```
                        prev = cur;
                        cur = cur->next;
                }


                else // If cur node is greater than pivot
                {
                        // Move cur node to next of tail, and change tail
                        if (prev != NULL)
                                prev->next = cur->next;
                        Node *tmp = cur->next;
                        cur->next = NULL;
                        tail->next = cur;
                        tail = cur;
                        cur = tmp;
                }
        }

        // If the pivot data is the smallest element in the current list,
        // pivot becomes the head
        if ((newHead) == NULL)
                (newHead) = pivot;

        // Update newEnd to the current last node
        (newEnd) = tail;

        // Return the pivot node
        return pivot;
}

//here the sorting happens exclusive of the end node
Node *quickSortRecur(Node *head, Node *end)
{
        // base condition
        if (!head || head == end)
                return head;

        Node *newHead = NULL, *newEnd = NULL;

        // Partition the list, newHead and newEnd will be updated
        // by the partition function
        Node *pivot = partition(head, end, newHead, newEnd);

        // If pivot is the smallest element - no need to recur for
        // the left part.
        if (newHead != pivot)
        {
                // Set the node before the pivot node as NULL
                Node *tmp = newHead;
                while (tmp->next != pivot)
                        tmp = tmp->next;
                tmp->next = NULL;

                // Recur for the list before pivot
                newHead = quickSortRecur(newHead, tmp);
```

```
                // Change next of last node of the left half to pivot
                tmp = getTail(newHead);
                tmp->next = pivot;
        }

        // Recur for the list after the pivot element
        pivot->next = quickSortRecur(pivot->next, newEnd);

        return newHead;
}

// The main function for quick sort. This is a wrapper over recursive
// function quickSortRecur()
void quickSort(Node *&headRef)
{
        (headRef) = quickSortRecur(headRef, getTail(headRef));
        return;
}
```

==========

****Radix sort Advanced**

```
void addTail(Ref &h, Ref &t, Ref p)
{
        if (h == NULL) {
                h = t = p;
                t->next = NULL;
        }
        else
        {
                p->next = NULL;
                t->next = p;
                t = p;
        }
}

int digit(int x, int k,int n)
{
        return ((x / (int)pow(10, n*k - n)) % (int)pow(10, n));
}

void radixSort_Advanced(Ref& h, Ref& t,int k)
{
        if (h == t) return;

        int max = h->data, d;

        //Tim max
        Ref q = h->next;
        while (q) {
                if (q->data > max)
                        max = q->data;
                q = q->next;
        }

        int m = (int)pow(10, k); //So lo can tao ra
        //Tinh so lan tach lo
        d = CountDigit(max);
```

```
        d = ceil((float)d / k);

        for (int i = 1; i <= d; i++)
        {
                Ref *head, *tail;
                head = (Ref*)malloc(sizeof(struct  node)*m);
                tail = (Ref*)malloc(sizeof(struct  node)*m);

                for (int t = 0; t < m; t++)
                {
                        head[t] = NULL;
                        tail[t] = head[t];
                }

                //Dua cac phan tu vao cac lo
                while (h)
                {
                        Ref p = h;
                        h = h->next;
                        p->next = NULL;
                        int lo = digit(p->data, i, k);
                        addTail(head[lo], tail[lo], p);
                }

                //Noi cac lo voi nhhau thanh danh sach voi h va t ban dau
                for (int j = 0; j < m; j++)
                {
                        if (h == NULL) {
                                if (head[j] != NULL) {
                                        h = head[j];
                                        t = tail[j];
                                        t->next = NULL;
                                }
                        }
                        else
                        {
                                if (head[j] != NULL) {
                                        t->next = head[j];
                                        t = tail[j];
                                        t->next = NULL;
                                }
                        }
                }
        }
}
```

==========

****Sparse table**

```
#include <iostream>
using namespace std;

struct Node
{
        int value;
        int row;
        int col;
        Node *hnext;
```

```
        Node *vnext;
};

struct RowHead
{
        int row;
        Node *node;
        RowHead *next;
};

struct ColHead
{
        int col;
        Node *node;
        ColHead *next;
};

struct SparseTable
{
        //int rows, cols;
        RowHead *row_ptr;
        ColHead *col_ptr;
};

Node* getNode(int value, int row, int col)
{
        Node *p = new Node;
        if (p == NULL)return NULL;
        p->value = value;
        p->row = row;
        p->col = col;
        p->hnext = NULL;
        p->vnext = NULL;
        return p;
}

RowHead* getRowHead(int row)
{
        RowHead *p = new RowHead;
        if (p == NULL)return NULL;
        p->row = row;
        p->node = NULL;
        p->next = NULL;
        return p;

}

ColHead* getColHead(int col)
{
        ColHead *p = new ColHead;
        if (p == NULL)return NULL;
        p->col = col;
        p->node = NULL;
        p->next = NULL;
        return p;
}
```

```
void add(SparseTable &a, int value, int row, int col)
{
        Node *node = getNode(value, row, col);

        if (a.row_ptr == NULL)
        {
                a.row_ptr = getRowHead(row);
                a.row_ptr->node = node;
        }
        else
        {
                RowHead *r = a.row_ptr;
                while (r != NULL)
                {
                        if (r->row == row)break;
                        r = r->next;
                }
                if (r == NULL)
                {
                        r = getRowHead(row);
                        r->node = node;

                        RowHead *p = a.row_ptr;
                        RowHead *prev = NULL;
                        while (p != NULL)
                        {
                                if (p->row > row)break;
                                prev = p;
                                p = p->next;
                        }
                        if (prev == NULL)
                        {
                                a.row_ptr = r;
                                r->next = p;
                        }
                        else
                        {
                                prev->next = r;
                                r->next = p;
                        }
                }
                else
                {
                        Node *prev = NULL, *p = r->node;
                        while (p != NULL)
                        {
                                if (p->col > col)break;
                                prev = p;
                                p = p->hnext;
                        }
                        if (prev == NULL)
                        {
                                r->node = node;
                                node->hnext = p;
                        }
```

```
                else
                {
                        prev->hnext = node;
                        node->hnext = p;
                }
        }
}

if (a.col_ptr == NULL)
{
        a.col_ptr = getColHead(col);
        a.col_ptr->node = node;
}
else
{
        ColHead *c = a.col_ptr;
        while (c != NULL)
        {
                if (c->col == col)break;
                c = c->next;
        }
        if (c == NULL)
        {
                c = getColHead(col);
                c->node = node;

                ColHead *p = a.col_ptr;
                ColHead *prev = NULL;
                while (p != NULL)
                {
                        if (p->col > col)break;
                        prev = p;
                        p = p->next;
                }
                if (prev == NULL)
                {
                        a.col_ptr = c;
                        c->next = p;
                }
                else
                {
                        prev->next = c;
                        c->next = p;
                }
        }
        else
        {
                Node *prev = NULL, *p = c->node;
                while (p != NULL)
                {
                        if (p->row > row)break;
                        prev = p;
                        p = p->vnext;
                }
                if (prev == NULL)
                {
```

```
                                c->node = node;
                                node->vnext = p;
                        }
                        else
                        {
                                prev->vnext = node;
                                node->vnext = p;
                        }
                }
        }
}

void print(SparseTable a)
{
        RowHead *x;
        ColHead *y;
        Node *i, *j;
        for (x = a.row_ptr; x != NULL; x = x->next)
        {
                for (y = a.col_ptr; y != NULL; y = y->next)
                {
                        i = x->node;
                        while (i != NULL)
                        {
                                j = y->node;
                                while (j != NULL)
                                {
                                        if (i == j)
                                        {
                                                cout << "\nSparseTable[" << x-
>row << "][" << y->col << "] = " << i->value;
                                        }
                                        j = j->vnext;
                                }
                                i = i->hnext;
                        }
                }
        }
}

int main()
{
        int Sparse_Matrix[4][5] =
        {
                { 0 , 0 , 3 , 0 , 4 },
                { 0 , 0 , 5 , 7 , 0 },
                { 0 , 0 , 0 , 0 , 0 },
                { 0 , 2 , 6 , 0 , 0 }
        };

        SparseTable a;
        a.row_ptr = NULL;
        a.col_ptr = NULL;

        for (int i = 0; i < 4; i++)
        {
```

```
                for (int j = 0; j < 5; j++)
                {
                        if (Sparse_Matrix[i][j] != 0)
                        {
                                add(a, Sparse_Matrix[i][j], i, j);
                        }
                }
        }

        print(a);

        system("pause");
        return 0;
}
```
==========

****Double linked list Josephus**

```
typedef struct Node * ref;
struct Node
{
        int key;
        Node* next;
};


Node* getNode(int k)
{
        Node* p = new Node;
        if (p == NULL)return NULL;
        p->key = k;
        p->next = NULL;
        return p;
}

int josephus(int n, int k)
{
        if (n == 1)
                return 1;
        else
                /* The position returned by josephus(n - 1, k) is adjusted
because the recursive call josephus(n - 1, k) considers the original position
k%n + 1 as position 1 */
                return (josephus(n - 1, k) + k - 1) % n + 1;
}

void getJosephusPosition(int m, int n)
{
        // tao 1 vong tu 1->n
        Node *head = getNode(1);
        Node *prev = head;
        for (int i = 2; i <= n; i++)
        {
                prev->next = getNode(i);
                prev = prev->next;
```

```
        }
        prev->next = head;  // noi duoi vao dau

        /* while only one node is left in the linked list*/
        Node *ptr1 = head, *ptr2 = head;
        while (ptr1->next != ptr1) // !=tail
        {
                // Find m-th node
                int count = 1;
                while (count != m)
                {
                        ptr2 = ptr1;
                        ptr1 = ptr1->next;
                        count++;
                }

                /* Remove the m-th node */
                ptr2->next = ptr1->next;

                Node* p = ptr1;
                delete p;
                p = NULL;

                ptr1 = ptr2->next;
        }

        printf("Last person left standing ""(Josephus Position) is %d\n ",
ptr1->key);
}
```

==========

****Topology Sort**

```
typedef struct leader* lref;
typedef struct trailer* tref;

struct leader
{
        int key;
        int count;
        lref next;
        tref trail;
};

struct trailer
{
        lref id;
        tref next;
};

lref addList(lref &head, lref &tail, int w, int &z)
{
        lref h = head;
        tail->key = w;

        while (h->key != w)
```

```
        {
                h = h->next;
        }

        if (h == tail)
        {
                tail = new leader;
                z++;
                h->count = 0;
                h->trail = NULL;
                h->next = tail;
        }
        return h;
}


int main()
{
        lref head, tail, p, q;
        tref t;
        head = new leader;
        tail = head;
        int z = 0;

        int x, y;
        cin >> x;
        while (x != 0) // dieu kien dung: x==0
        {
                cin >> y;
                p = addList(head, tail, x, z);
                q = addList(head, tail, y, z);
                t = new trailer; t->id = q; t->next = p->trail;
                p->trail = t; q->count++;
                cin >> x;
        }

        //in danh
sach************************************************************
        p = head; head = NULL;
        while (p != tail)
        {
                q = p;
                p = p->next;
                if (q->count == 0)
                {
                        q->next = head;
                        head = q;
                }
        }

        q = head;
        while (q)
        {
                cout << q->key << "   ";
                z--;
                t = q->trail; q = q->next;
```

```
                while (t)
                {
                        p = t->id; p->count--;
                        if (p->count == 0)
                        {
                                p->next = q;
                                q = p;
                        }
                        t = t->next;
                }
        }

        system("pause");
        return 0;
}
```
==========

**Binay Search Tree (BST)

```
struct Node
{
	int key;
	Node *left;
	Node *right;
};

Node* getNode(int k)
{
	Node *p = new Node;
	if (p == NULL)return NULL;
	p->key = k;
	p->left = NULL;
	p->right = NULL;
	return p;
}

//tim kiem
Node* search(Node *r, int k)
{
	if (r == NULL)return NULL;
	Node *p = r;
	while (p)
	{
		if (p->key == k)return p;
		if (p->key < k)p = p->right;
		else p = p->left;
	}
	return p;
}

//tim them : de quy
void them(Node *&r, int k)
{
	if (r == NULL)r = getNode(k);
	else
	{
		if (r->key < k)
		{
			them(r->right, k);
		}
		else if (r->key > k)
		{
			them(r->left, k);
		}
		else cout << "Da co!";
	}
}

//tim them : khu de quy
void them2(Node *&r, int k)
{
	Node *p2 = r;
	Node *p1 = r->right;
	int d = 1;
```

```
        while (p1&&d != 0)
        {
                if (p1->key < k)
                {
                        p2 = p1; p1 = p1->right; d = 1;
                }
                else if (p1->key > k)
                {
                        p2 = p1; p1 = p1->left; d = -1;
                }
                else d = 0;
        }
        if (d == 0)cout << "Da co!";
        else
        {
                p1 = getNode(k);
                if (d == 1)p2->right = p1;
                else p2->left = p1;
        }
}

//duyet giua
void duyetGiua(Node *r)
{
        if (r)
        {
                duyetGiua(r->left);
                cout << r->key << "   ";
                duyetGiua(r->right);
        }
}

========================================================================
**Cay Nhi Phan Can Bang Hoan Toan
struct Node
{
        int key;
        Node *left;
        Node *right;
};

Node* getNode(int k)
{
        Node *p = new Node;
        if (p == NULL)return NULL;
        p->key = k;
        p->left = NULL;
        p->right = NULL;
        return p;
}

//xay dung
Node* tree(Node *r, int n) //r: con tro tro den nut goc, n: so nut
{
        if (n == 0)return NULL;
```

```
        int nl = n / 2, nr = n - nl - 1;

        int k;
        cin >> k;
        Node *p = getNode(k);
        p->left = tree(r, nl);
        p->right = tree(r, nr);
        return p;
}

//duyet truoc
void duyetTruoc(Node *r)
{
        if (r)
        {
                cout << r->key << "   ";
                duyetTruoc(r->left);
                duyetTruoc(r->right);
        }
}

//duyet giua
void duyetGiua(Node *r)
{
        if (r)
        {
                duyetGiua(r->left);
                cout << r->key << "   ";
                duyetGiua(r->right);
        }
}

//duyet sau
void duyetSau(Node *r)
{
        if (r)
        {
                duyetSau(r->left);
                duyetSau(r->right);
                cout << r->key << "   ";
        }
}

//************************************************
//khu de quy

//duyet truoc
void preOrder(Node *r)
{
        Node *p = r;
        stack<Node*>s;
        if (p)
        {
                s.push(p);
                while (!s.empty())
                {
```

```
                        p = s.top();
                        s.pop();
                        cout << p->key << "   ";
                        if (p->right)s.push(p->right);
                        if (p->left)s.push(p->left);
                }
        }
}

//duyet sau
void postOrder(Node *r)
{
        Node *p = r, *q = r;
        stack<Node*>s;
        while (p)
        {
                for (; p->left; p = p->left)
                {
                        s.push(p);
                }
                while (p->right == NULL || p->right == q)
                {
                        cout << p->key << "   "; q = p;
                        if (s.empty())return;
                        p = s.top(); s.pop();
                }
                s.push(p);
                p = p->right;
        }
}

//duyet sau 2
void postOrder2(Node *r)
{
        Node *p = r;
        stack<Node*>s;
        stack<Node*>temp;
        if (p)
        {
                s.push(p);
                while (!s.empty())
                {
                        p = s.top();
                        s.pop();
                        temp.push(p);

                        if (p->left)s.push(p->left);
                        if (p->right)s.push(p->right);
                }
        }
        while (!temp.empty())
        {
                Node *q = temp.top(); temp.pop();
                cout << q->key << "   ";
        }
}
```

```
//duyet giua
void inOrder(Node *r)
{
	Node *p = r;
	stack<Node*>s;
	while (p)
	{
		while (p)
		{
			if (p->right)
			{
				s.push(p->right);
			}
			s.push(p);
			p = p->left;
		}
		p = s.top(); s.pop();
		while (!s.empty() && p->right == NULL)
		{
			cout << p->key << "   ";
			p = s.top(); s.pop();
		}
		cout << p->key << "   ";
		if (!s.empty())
		{
			p = s.top(); s.pop();
		}
		else p = NULL;
	}
}
```

==========

*AVL

```
//Khai bao struct
typedef struct tagNodeAVL
{
int key;
struct tagNodeAVL *left, *right;
char balIndex;
} NodeAVL;

//=============================================
NodeAVL* createNodeAVL(int key)
{
NodeAVL *p = new NodeAVL;
p->key = key;
p->left = p->right = NULL;
p->balIndex = EH;
return p;
```

```
}
//=========================================
void LNR(NodeAVL *root)
{
if(root!=NULL)
{
LNR(root->left);
cout<<root->key<<" ";
LNR(root->right);
}
}
//=========================================
void NLR(NodeAVL *root)
{
if(root!=NULL)
{
cout<<root->key<<" ";
NLR(root->left);
NLR(root->right);
}
}
//=========================================
void LRN(NodeAVL *root)
{
if(root!=NULL)
{
LRN(root->left);
LRN(root->right);
cout<<root->key<<" ";
}
}
//=========================================
void NRL(NodeAVL *root)
{
if(root!=NULL)
{
cout<<root->key<<" ";
NRL(root->right);
NRL(root->left);
}
}
```

```
//=============================================
void RNL(NodeAVL *root)
{
if(root!=NULL)
{
RNL(root->right);
cout<<root->key<<" ";
RNL(root->left);
}
}
//=============================================
void RLN(NodeAVL *root)
{
if(root!=NULL)
{
RLN(root->right);
RLN(root->left);
cout<<root->key<<" ";
}
}
//=============================================
/*
T bi mat can bang LL, LB
sau khi can bang T se bi thay bang T1 (left cua T)
*/
void rotateLL(NodeAVL* &T)
{
NodeAVL* T1 = T->left;
T->left = T1->right;
T1->right = T;
switch(T1->balIndex)
{
//truong hop nut T lech LL, khi do T1 lech ve ben trai
//nen balIndex cua no la -1 (LH)
case LH: T1->balIndex = T->balIndex = EH; break;
//truong hop nut T lech LB, khi do T1 can bang, nen
//balIndex cua no la 0 (EH)
case EH:
T1->balIndex = RH;
T->balIndex = LH;
break;
```

```
}

T = T1;
}
//=============================================
void rotateRR(NodeAVL* &T)
{
NodeAVL* T1 = T->right;
T->right = T1->left;
T1->left = T;
switch(T1->balIndex)
{
//truong hop nut T lech LL, khi do T1 lech ve ben trai
//nen balIndex cua no la -1 (LH)
case RH: T1->balIndex = T->balIndex = EH; break;

//truong hop nut T lech LB, khi do T1 can bang, nen
//balIndex cua no la 0 (EH)
case EH:
T1->balIndex = LH;
T->balIndex = RH;
break;
}

T = T1;
}
//=============================================
/*
T bi mat can bang LR, sau khi quay T2 se thay cho T
Thuc hien quay kep
Ban dau T co balIndex = -1 (LH), T1 co balIndex = 1 (RH)
*/
void rotateLR(NodeAVL* &T)//T bi mat can bang LR
{
NodeAVL* T1 = T->left;
NodeAVL* T2 = T1->right;

T1->right = T2->left;
T2->left = T1;

T->left = T2->right;
```

```
T2->right = T;

switch(T2->balIndex)
{
//truong hop T2 lech ben trai
case LH: T->balIndex = RH;
T1->balIndex = EH;
break;
case EH: T->balIndex = EH;
T1->balIndex = EH;
break;

case RH: T->balIndex = EH;
T1->balIndex = LH;
break;
}
T2->balIndex = EH;
T=T2;
}
//============================================
void rotateRL(NodeAVL *&T)
{
NodeAVL* T1 = T->right;
NodeAVL* T2 = T1->left;
T1->left = T2->right;
T2->right = T1;
T->right= T2->left;
T2->left = T;
switch(T2->balIndex)
{
case RH:T->balIndex = LH;
T1->balIndex = EH;
break;
case EH:T->balIndex = EH;
T1->balIndex = EH;
break;
case LH:T->balIndex = EH;
T1->balIndex = RH;
break;
}
T2->balIndex = EH;
```

```
T = T2;
}
//============================================
//ham can bang cay T lech ve ben trai.T co balIndex=-1(LH)
int balanceLeft(NodeAVL *&T)
{
NodeAVL* T1=T->left;
switch(T1->balIndex)
{
//T bi lech LL
case LH:rotateLL(T);
return 2;
break;

//T bi lech LB
case EH:rotateLL(T);
return 1;
break;

//T bi lech LR
case RH:rotateLR(T);
return 2;
break;
}
return 0;
}
//============================================
//ham can bang cay T lech ve ben phai. T co balIndex=1(RH)
int balanceRight(NodeAVL *&T)
{
NodeAVL* T1=T->right;
switch(T1->balIndex)
{
//T bi lech LL
case LH:rotateRL(T);
return 2;
break;

//T bi lech LB
case EH:rotateRR(T);
return 1;
```

```
break;

//T bi lech LR
case RH:rotateRR(T);
return 2;
break;
}
return 0;
}
//============================================
int insertNode(NodeAVL* &T, int x)
{
int res;
if(T)
{
if(T->key==x) return 0;//da co
if(T->key>x)
{
res=insertNode(T->left, x);
if(res<2) return res;
switch(T->balIndex)
{
case RH:
T->balIndex=EH;
return 1;
case EH:
T->balIndex=LH;
return 2;
case LH:
balanceLeft(T);
return 1;
}
}
else
{
res=insertNode(T->right,x);
if(res<2) return res;
switch(T->balIndex)
{
case LH:
T->balIndex=EH;
```

```
return 1;
case EH:
T->balIndex=RH;
return 2;
case RH:
balanceRight(T);
return 1;
}
}
}
T=(NodeAVL*) new NodeAVL;
if(T==NULL) return -1;
T->key=x;
T->balIndex=EH;
T->left=T->right=NULL;
return 2;
}
//=============================================
int searchStandFor(NodeAVL* &p,NodeAVL* &q)
{
int res;
if(q->left)
{
res=searchStandFor(p,q->left);
if(res<2) return res;
switch(q->balIndex)
{
case LH:
q->balIndex=EH;
return 2;
case EH:
q->balIndex=RH;
return 1;
case RH:
return balanceRight(q);
}
}
else
{
p->key=q->key;
p=q;
```

```
q=q->right;
}
return 2;
}
//=============================================
int deleteNode(NodeAVL* &T, int x)
{
int res;
if(T==NULL) return 0;
if(T->key>x)
{
res=deleteNode(T->left,x);
if(res<2) return res;
switch(T->balIndex)
{
case LH:
T->balIndex=EH;
return 2;
case EH:
T->balIndex=RH;
return 1;
case RH:
return balanceRight(T);
}
}
if(T->key<x)
{
res=deleteNode(T->right,x);
if(res<2) return res;
switch(T->balIndex)
{
case RH:
T->balIndex=EH;
return 2;
case EH:
T->balIndex=LH;
return 1;
case LH:
return balanceLeft(T);
}
}
```

```
else
{
NodeAVL *p=T;
if(T->left==NULL)
{
T=T->right;
res=2;
}
else
{
if(T->right==NULL)
{
T=T->left;
res=2;
}
else
{
res=searchStandFor(p,T->right);
if(res<2) return res;
switch(T->balIndex)
{
case RH:
T->balIndex=EH;
return 2;
case EH:
T->balIndex=LH;
return 1;
case LH:
return balanceLeft(T);
}
}
}
delete p;
return res;
}
return res;
}
//============================================
void createTreeFromTextFile(NodeAVL* &root)
{
ifstream in;
```

```
int n;//Chua so phan tu cua cay
char* filename = "filein.txt";

in.open(filename);
in>>n;
for(int i=1; i<=n; i++)
{
int x;//Chua key
in>>x;
insertNode(root, x);
}
}
//============================================
int searchNode1DQ(NodeAVL *root, int x)//Tra ve 1:tim thay nut chua khoa, 0:khong tim thay
nut chua khoa
{
if(root==NULL) return 0;
if(root->key==x) return 1;
if(root->key > x)
return searchNode1DQ(root->left, x);
else
return searchNode1DQ(root->right, x);
}
//============================================
int searchNode1KDQ(NodeAVL *root,int x)//Tra ve 1:tim thay nut chua khoa, 0:khong tim thay
nut chua khoa
{
NodeAVL *p = root;
while(p!=NULL && p->key!=x)
{
if(p->key > x)
p=p->left;
else
p=p->right;
}

if(p==NULL)
return 0;
else
return 1;
}
```

```
//============================================
NodeAVL *searchNode2DQ(NodeAVL *root, int x)
{
if(root==NULL) return NULL;
if(root->key==x) return root;
if(root->key > x)
return searchNode2DQ(root->left, x);
else
return searchNode2DQ(root->right, x);
}
//============================================
NodeAVL *searchNode2KDQ(NodeAVL *root,int x)
{
NodeAVL *p = root;
while(p!=NULL && p->key!=x)
{
if(p->key > x)
p=p->left;
else
p=p->right;
}
return p;
}
//============================================
int MucNodeChuaKhoaDQ(NodeAVL *root, int x)//Tim muc cua nut chua khoa
{
if(root==NULL) return 0;
if(root->key==x) return 0;
if(root->key > x)
return 1 + MucNodeChuaKhoaDQ(root->left, x);
else
return 1 + MucNodeChuaKhoaDQ(root->right, x);
}
//============================================
int timMucNodeChuaKhoaDQ(NodeAVL *root, int x)
{
if(searchNode1DQ(root, x))//Kiem tra co nut chua khoa trong cay hay khong
return MucNodeChuaKhoaDQ(root, x);//Tim muc cua nut chua khoa
else
return -1;//Khong co nut chua khoa trong cay
}
```

```
//================================================
int timMucNodeChuaKhoaKDQ(NodeAVL *root, int x)
{
if(searchNode1KDQ(root, x))
{
NodeAVL *p = root;
int Muc = 0;
while(p->key!=x)
{
if(p->key > x)
{
p=p->left;
Muc++;
}
else
{
p=p->right;
Muc++;
}
}
return Muc;
}
else
return -1;//Khong co nut chua khoa trong cay
}
//================================================
void searchKeyMax(NodeAVL *root)
{
NodeAVL *p = root;
if(p==NULL)
cout<<"Cay rong! Khong co so lon nhat!"<<endl;
else
{
while(p->right!=NULL)
p=p->right;
cout<<"So lon nhat tren cay la "<<p->key<<endl;
}
}
//================================================
void searchKeyMin(NodeAVL *root)
{
```

```
NodeAVL *p = root;
if(p==NULL)
cout<<"Cay rong! Khong co so nho nhat!"<<endl;
else
{
while(p->left!=NULL)
p=p->left;
cout<<"So nho nhat tren cay la "<<p->key<<endl;
}
}
//=======================================================
int demNodeLa(NodeAVL *root)
{
if(root==NULL)
return 0;
if(root->left==NULL && root->right==NULL)
return 1;
else
return demNodeLa(root->left) + demNodeLa(root->right);
}
//=======================================================
void xuatNodeLa(NodeAVL *root)
{
if(root==NULL)
return;
if(root->left==NULL && root->right==NULL)
cout<<root->key<<" ";
else
{
xuatNodeLa(root->left);
xuatNodeLa(root->right);
}
}
//=======================================================
int demNodeKhongLa(NodeAVL *root)
{
if (root==NULL) return 0;
if (root->left==NULL && root->right==NULL)
return 0;
else
return 1+demNodeKhongLa(root->left)+demNodeKhongLa(root->right);
```

```
}
//=========================================================
void xuatNodeKhongLa(NodeAVL *root)
{
if(root==NULL)
return;
if(root->left==NULL && root->right==NULL)
return;
else
{
cout<<root->key<<" ";
xuatNodeKhongLa(root->left);
xuatNodeKhongLa(root->right);
}
}
//=========================================================
int demNodeTrong(NodeAVL *root)
{
if (root==NULL) return 0;
if (root->left==NULL && root->right==NULL)
return 0;
else
return 1+demNodeTrong(root->left)+demNodeTrong(root->right);
}
//=========================================================
int DoDai(NodeAVL *root, int x)
{
if(searchNode1DQ(root,x))
{
if(root->key==x) return 0;
else if(root->key>x) return 1 + DoDai(root->left,x);
else return 1 + DoDai(root->right,x);
}
else
return -1;//Khong co nut chua khoa trong cay
}
//=========================================================
int Max(int x,int y)
{
return(x>y)? x:y;
}
```

```
//============================================================
int Height(NodeAVL* root)//Tra ve -1:cay rong, so khong am(>=0):cay khong rong
{
if(root == NULL)
return -1;
return Max(1 + Height(root->left),1 + Height(root->right));
}
//============================================================
int MotNodeCon(NodeAVL *root)
{
if(root==NULL) return 0;
if((root->left==NULL && root->right!=NULL)||(root->left!=NULL && root->right==NULL))
return 1;
else return MotNodeCon(root->left) + MotNodeCon(root->right);
}
//============================================================
int HaiNodeCon(NodeAVL *root)
{
if(root==NULL) return 0;
if((root->left==NULL && root->right!=NULL)||(root->left!=NULL && root->right==NULL)||(root-
>left==NULL && root->right==NULL))
return 0;
else return 1 + HaiNodeCon(root->left) + HaiNodeCon(root->right);
}
//============================================================
void xuatKeyNodeCungMuc(NodeAVL *root, int k)
{
if(root==NULL)
return;
if(k==0)
cout<<root->key<<" ";
else
{
k--;
xuatKeyNodeCungMuc(root->left, k);
xuatKeyNodeCungMuc(root->right, k);
}
}
//============================================================
void GiaTriChan(NodeAVL *root)
{
```

```
if(root!=NULL)
{
if(root->key%2==0) cout<<root->key<<" ";
GiaTriChan(root->left);
GiaTriChan(root->right);
}
}
//=============================================================
void LonHonXNhoHonY(NodeAVL *root,int x,int y)
{
if(root!=NULL)
{
if(root->key>x && root->key<y)
cout<<root->key<<" ";
LonHonXNhoHonY(root->left,x,y);
LonHonXNhoHonY(root->right,x,y);
}
}
//=============================================================
int SoHoanThien(int n)
{
int tong=0;
for(int i=1;i<=n;i++)
{
if(n%i==0) tong=tong+i;
}
if(tong==2*n && tong!=0) return 1;
else return 0;
}
//=============================================================
void XuatSoHoanThien(NodeAVL *root)
{
if(root==NULL) return;
if(SoHoanThien(root->key)==1)
cout<<root->key<<" ";
XuatSoHoanThien(root->left);
XuatSoHoanThien(root->right);
}
//=============================================================
void XuatNodeDuoiMucK(NodeAVL *root,int k)
{
```

```
for(int i=k+1; i<=Height(root); i++)
xuatKeyNodeCungMuc(root, i);
}
//============================================================
void XuatNodeTang0DenH1(NodeAVL *root)
{
int h=Height(root);
for(int i=0;i<h;i++)
{
cout<<"Muc "<<i<<" : ";
xuatKeyNodeCungMuc(root,i);
cout<<endl;
}
}
//============================================================
int SoNodeChan(NodeAVL *root)
{
if(root==NULL) return 0;
int a=SoNodeChan(root->left);
int b=SoNodeChan(root->right);
if(root->key%2==0) return 1+a+b;
return a+b;
}
//============================================================
int SoNguyenTo(int n)
{
if(n < 2)
return 0;
for(int i=2; i<n;i++)
{
if(n%i==0) return 0;
}
return 1;
}
//============================================================
int DemNodeNT1Con(NodeAVL *root)
{
if(root==NULL) return 0;
int a=DemNodeNT1Con(root->left);
int b=DemNodeNT1Con(root->right);
if(SoNguyenTo(root->key)==1 &&((root->left!=NULL && root->right==NULL)||
```

```
(root->left==NULL && root->right!=NULL))) return 1+a+b;
return a+b;
}
//============================================================
int SoChinhPhuong(int n)
{
for(int i=1;i<=n;i++)
if(n==i*i) return 1;
return 0;
}
//============================================================
int DemNodeCP2Con(NodeAVL *root)
{
if(root==NULL) return 0;
int a=DemNodeCP2Con(root->left);
int b=DemNodeCP2Con(root->right);
if(SoChinhPhuong(root->key)==1 && root->left!=NULL && root->right!=NULL)
return 1+a+b;
return a+b;
}
//============================================================
int DemNodeTangK(NodeAVL *root,int k)
{
if(root==NULL) return 0;
int a=DemNodeTangK(root->left,k-1);
int b=DemNodeTangK(root->right,k-1);
if(k==0) return 1+a+b;
return a+b;
}
//============================================================
int DemNodeThapHonTangK(NodeAVL *root,int k)
{
int tong=0;
for(int i=k+1; i<=Height(root); i++)
tong+=DemNodeTangK(root,i);
return tong;
}
//============================================================
int DemNodeCaoHonTangK(NodeAVL *root,int k)
{
int tong=0;
```

```
for(int i=0; i<k; i++)
tong+=DemNodeTangK(root,i);
return tong;
}
//============================================================
int TongNode(NodeAVL *root)
{
if(root==NULL) return 0;
else return root->key+TongNode(root->left)+TongNode(root->right);
}
//============================================================
int TongNodeLa(NodeAVL *root)
{
if(root==NULL) return 0;
if(root->left==NULL && root->right==NULL) return root->key;
else return TongNodeLa(root->left)+ TongNodeLa(root->right);
}
//============================================================
int TongNodeMotCon(NodeAVL *root)
{
if(root==NULL) return 0;
int a=TongNodeMotCon(root->left);
int b=TongNodeMotCon(root->right);
if((root->left==NULL && root->right!=NULL)||(root->left!=NULL && root->right==NULL))
return root->key+a+b;
return a+b;
}
//============================================================
int TongNodeHaiCon(NodeAVL *root)
{
if(root==NULL) return 0;
int a=TongNodeHaiCon(root->left);
int b=TongNodeHaiCon(root->right);
if(root->left!=NULL && root->right!=NULL)
return root->key+a+b;
return a+b;
}
//============================================================
int TongNodeGiaTriLe(NodeAVL *root)
{
if(root==NULL) return 0;
```

```
int a=TongNodeGiaTriLe(root->left);
int b=TongNodeGiaTriLe(root->right);
if(root->key%2==1) return root->key+a+b;
return a+b;
}
//============================================================
int TongNodeGiaTriChan(NodeAVL *root)
{
if(root==NULL) return 0;
int a=TongNodeGiaTriChan(root->left);
int b=TongNodeGiaTriChan(root->right);
if(root->key%2==0) return root->key+a+b;
return a+b;
}
//============================================================
int TongNodeNT1Con(NodeAVL *root)
{
if(root==NULL) return 0;
int a=TongNodeNT1Con(root->left);
int b=TongNodeNT1Con(root->right);
if(SoNguyenTo(root->key)==1 &&((root->left!=NULL && root->right==NULL)||
(root->left==NULL && root->right!=NULL)))
return root->key + a + b;
return a + b;
}
//============================================================
int TongNodeCP2Con(NodeAVL *root)
{
if(root==NULL) return 0;
int a=TongNodeCP2Con(root->left);
int b=TongNodeCP2Con(root->right);
if(SoChinhPhuong(root->key)==1 && root->left!=NULL && root->right!=NULL)
return root->key + a + b;
return a + b;
}
//============================================================
int KeyMax(NodeAVL *root)
{
if(root==NULL)
return 0;
if(root->left==NULL && root->right==NULL)
```

```
return root->key;
else
return Max(Max(root->key, KeyMax(root->left)), KeyMax(root->right));
}
//============================================================
int isBStree(NodeAVL *root)
{
if(root==NULL) return 1;
if(root->left!=NULL && root->right==NULL && root->key<KeyMax(root->left))
return 0;
if(root->left==NULL && root->right!=NULL && root->key>KeyMax(root->right))
return 0;
if(root->left!=NULL && root->right!=NULL && (root->key<KeyMax(root->left) ||
root->key>KeyMax(root->right)))
return 0;
return isBStree(root->left)&&isBStree(root->right);
}
//============================================================
void testBStree(NodeAVL *root)
{
if(isBStree(root)==1)
cout<<"Cay da nhap la cay nhi phan tim kiem!"<<endl;
else
cout<<"Cay da nhap khong phai cay nhi phan tim kiem!"<<endl;
}
//============================================================
int isCanBang(NodeAVL *root)
{
if(root==NULL) return 1;
if(abs(Height(root->left) - Height(root->right)) > 1)
return 0;
return isCanBang(root->left)&&isCanBang(root->right);
}
//============================================================
int isAVLtree(NodeAVL *root)
{
if(root==NULL) return 1;
if(isBStree(root)==1 && isCanBang(root)==1)
return 1;
return 0;
}
```

```
//============================================================
void testAVLtree(NodeAVL *root)
{
if(isAVLtree(root)==1)
cout<<"Cay da nhap la cay nhi phan tim kiem can bang (cay AVL)!"<<endl;
else
cout<<"Cay da nhap khong phai cay nhi phan tim kiem can bang (cay AVL)!"<<endl;
}
//============================================================
int SoNode(NodeAVL *root)
{
if(root==NULL) return 0;
else return 1 + SoNode(root->left) + SoNode(root->right);
}
//============================================================
int isCBHT(NodeAVL *root)
{
if(root==NULL) return 1;
if(abs(SoNode(root->left) - SoNode(root->right)) > 1)
return 0;
return isCBHT(root->left)&&isCBHT(root->right);
}
//============================================================
int isNPCBHTtree(NodeAVL *root)
{
if(root==NULL) return 1;
if(isBStree(root)==1 && isCBHT(root)==1)
return 1;
return 0;
}
//============================================================
void testNPCBHTtree(NodeAVL *root)
{
if(isNPCBHTtree(root)==1)
cout<<"Cay da nhap la cay nhi phan tim kiem can bang hoan toan!"<<endl;
else
cout<<"Cay da nhap khong phai cay nhi phan tim kiem can bang hoan toan!"<<endl;
}
```

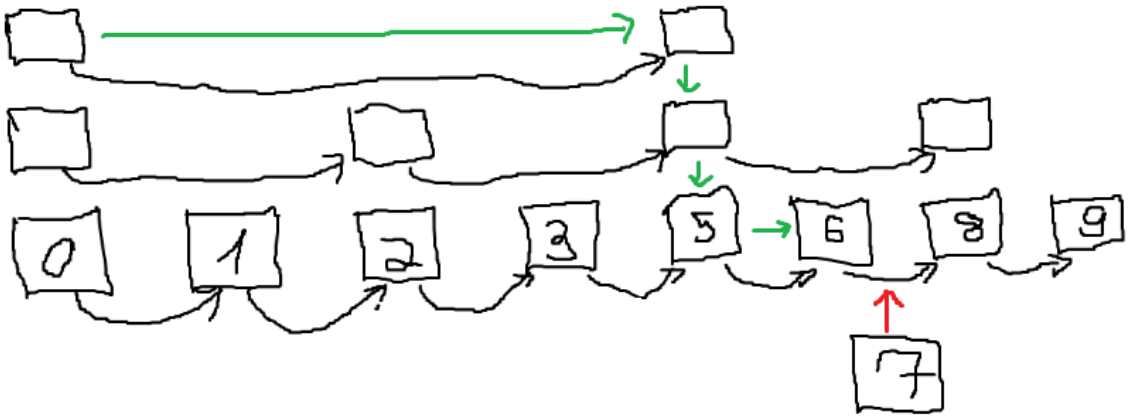
nhiên lại dễ cài đặt hơn rất nhiều.

Ý tưởng Skip Lists

Skip Lists là một phiên bản nâng cấp của Sorted Linked Lists. Ta hãy bắt đầu với một ví dụ về Sorted Linked List chứa 8 số và nghĩ cách cải thiện vấn đề của nó.

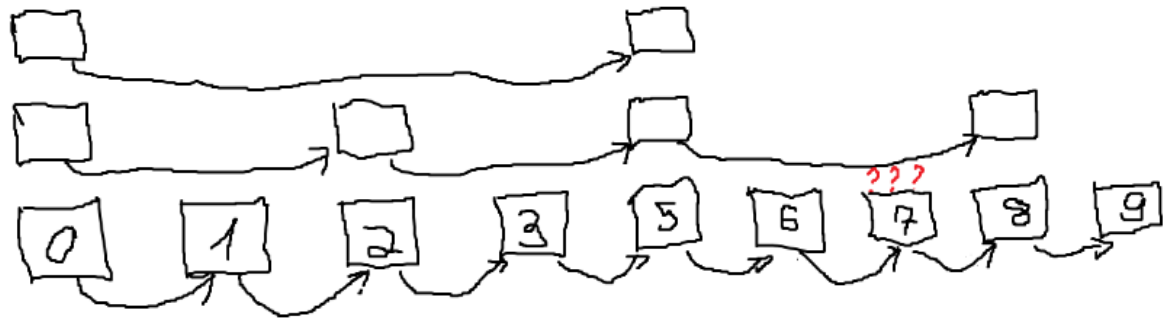
Sorted Linked List có ưu điểm lớn khi thao tác chèn xóa chỉ mất *[Math Processing Error]O(1)* (ta chỉ việc chỉnh sửa liên kết giữa phần tử được chèn/xóa và các phần tử đằng trước/sau). Tuy nhiên thao tác tìm kiếm lại mất *[Math Processing Error]O(N)* do phải duyệt từ đầu đến cuối.

Một ý tưởng để cân bằng điều này là ta thêm nhiều tầng liên kết, cứ lên một tầng số liên kết lại giảm còn một nửa. Khi tìm phần tử, ta sẽ duyệt từ trái sang phải nhưng sẽ nhảy xa hơn nhờ những liên kết trên các tầng cao, khi nào không nhảy được mới xuống tầng thấp hơn. Ý tưởng này khá giống với phương pháp nhảy lên tổ tiên thứ *[Math Processing Error]2K* khi tìm Lowest Common Ancestor (LCA).



Trong hình trên, để tìm số *[Math Processing Error]7*, ta sẽ nhảy thẳng từ *[Math Processing Error]0* đến *[Math Processing Error]5*bằng liên kết trên tầng thứ ba, sau đó nhảy từ *[Math Processing Error]5* đến *[Math*

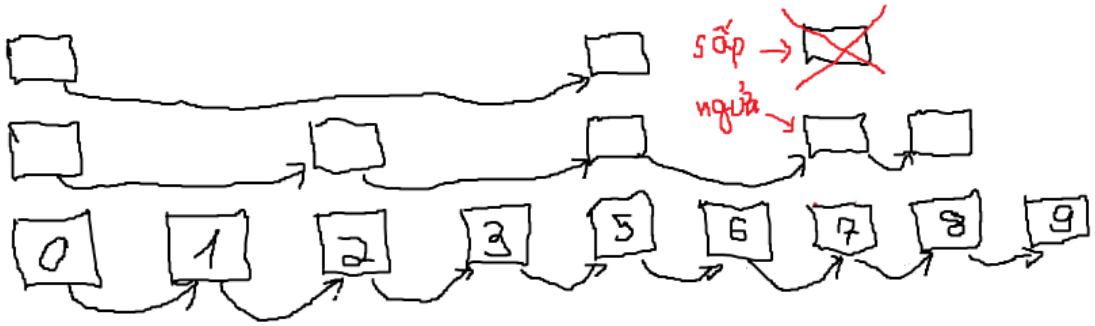
Processing Error]6 bằng liên kết trên tầng thứ nhất. Ta tìm được *[Math Processing Error]6* là số gần nhất với *[Math Processing Error]7*.
Với cấu trúc này, ta có thể thực hiện thao tác tìm trong *[Math Processing Error]O(log(N))*. Tuy nhiên việc chèn và xóa một phần tử vào sẽ làm thay đổi cấu trúc này. Chẳng hạn nếu ta chèn số *[Math Processing Error]7*:



Như hình trên, cấu trúc của ta không còn "chuẩn", có nghĩa là chính xác tầng thứ nhất liên kết cách *[Math Processing Error]20*, tầng thứ hai liên kết cách *[Math Processing Error]21*, tầng thứ ba liên kết cách *[Math Processing Error]22*, ... Tuy nhiên, với cấu trúc như hình trên vẫn chạy tốt - chỉ có điều ở mỗi tầng ta có thể phải nhảy nhiều hơn một lần (chẳng hạn, muốn tìm số *[Math Processing Error]7*, ở tầng thứ nhất ta phải nhảy đến hai lần *[Math Processing Error]5* ~> *[Math Processing Error]6* ~> *[Math Processing Error]7*).

Từ đó ta có nhận xét sau: Các liên kết trên mỗi tầng không nhất thiết phải chuẩn, tuy nhiên, nếu các độ dài giữa các liên kết xấp xỉ nhau và số liên kết ở tầng trên xấp xỉ bằng nửa số liên kết ở tầng dưới, thuật toán tìm kiếm vẫn chạy tốt và không mất quá nhiều lần nhảy ở mỗi tầng. Ta sẽ duy trì cấu trúc này bằng kĩ thuật tung đồng xu ngẫu nhiên:

Mỗi lần chèn một nút vào, đầu tiên ta xây dựng liên kết ở tầng thứ nhất cho nó. Sau đó ta tung đồng xu, nếu ngửa thì ta xây dựng liên kết ở tầng trên và tiếp tục tung đồng xu, còn nếu sấp ta dừng việc xây dựng liên kết lại.



Đây chính là Skip Lists - một cấu trúc dữ liệu được xây dựng bằng nhiều tầng Sorted Linked List được xây dựng một cách ngẫu nhiên, trong đó tầng cao chứa những bước nhảy dài hơn và tầng thấp chứa những bước nhảy

ngắn hơn. Skip Lists cho phép ta thực hiện thao tác tìm kiếm với độ phức tạp xấp xỉ *[Math Processing Error]O(Log(N))*.

So sánh các cấu trúc dữ liệu

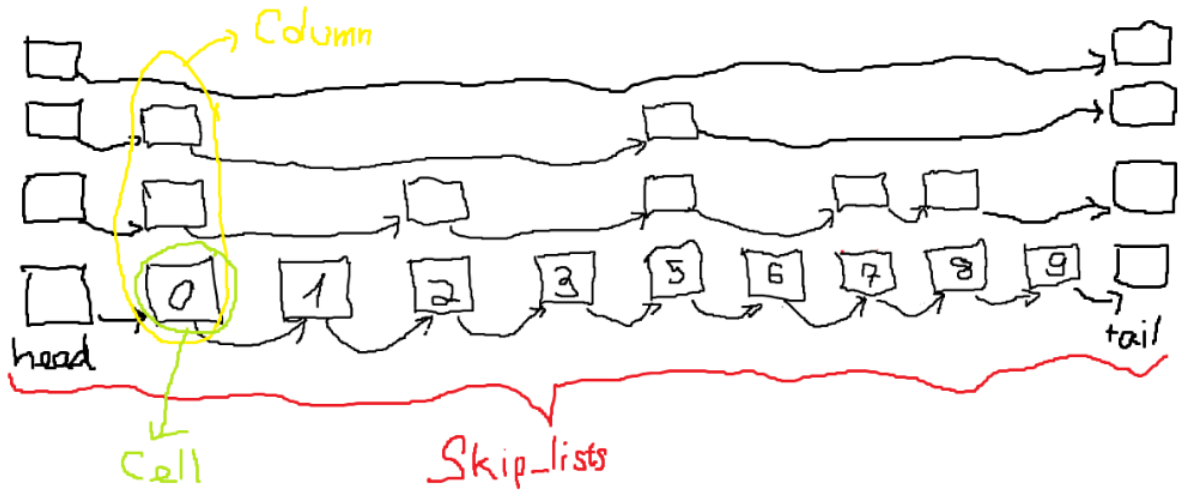
	Sorted Array	Sorted Linked List	Binary Search Tree	Red-Black Tree (a Balanced BST)	Skip Lists
Bộ nhớ	O(N)	O(N)	O(N)	O(N)	O(N)
Chèn	O(N)	O(1)	O(log(N))	O(log(N))	O(log(N))
Xóa	O(N)	O(1)	O(log(N))	O(log(N))	O(log(N))
Tìm	O(log(N))	O(N)	O(log(N))	O(log(N))	O(log(N))
So sánh	* Cực dễ code * Chèn xóa chậm * ĐPT ổn định	* Cực dễ code * Tìm chậm * ĐPT ổn định	* Khó code * Nhanh * ĐPT ngẫu nhiên (sinh test chết được)	* Cực khó code * Nhanh * ĐPT ổn định (tùy loại)	* Dễ code * Nhanh * ĐPT ngẫu nhiên (chưa có cách sinh test chết)

Hướng dẫn chi tiết

Học phải đi đôi với hành. Cách hiểu lý thuyết nhanh nhất là đập ngay vào bài tập. Ta sẽ đi chi tiết vào cách sử dụng Skip Lists để giải bài CPPSET. Bạn hãy đọc đề và ngẫm nghĩ một lúc trước khi đọc tiếp bài viết này. Bài giải ở dưới được code bằng ngôn ngữ C++98.

CPPSET, đúng như tên gọi của nó, bạn có thể AC trong một nốt nhạc nếu sử dụng std::set của C++, một container đã được code sẵn bằng Reb-Black Tree (một loại Balanced Binary Search Tree) để thực hiện bài toán cơ bản nêu ở đầu. Để luyện tập, ta sẽ tự code một cái set "dỏm" bằng Skip Lists.

Trước tiên ta cần xây dựng các struct biểu diễn Skip Lists. Ta sẽ có 3 struct: `SkipLists`, `Column`, `Cell`. `SkipLists` là một danh sách các `Column` liên kết với nhau. `Column` là một cột gồm các `Cell`, biểu diễn cho cột liên kết của một phần tử trong set của ta với các phần tử đằng trước và đằng sau. `Cell` là một liên kết cơ bản nhất trên một tầng của `Column`, chứa hai liên kết đến `Column` đằng trước và đằng sau. Để cho dễ hiểu, bạn hãy xem hình dưới.




```
struct SkipLists {
```

```
    static const int MAX_LEVEL = 20; // Giới hạn số tầng, nên chọn một số khoảng log(N)
    Column *head, *tail; // thêm 2 cột không có giá trị vào đầu và cuối để dễ xử lí
};

struct Column {
    int value;
    vector<Cell> cells;
};

struct Cell {
    Column *previous_column, *next_column; // Mỗi Cell có hai liên kết đến Column đằng trước và đằng sau (không giống trong hình chỉ vẽ một liên kết về đằng sau cho đơn giản)
};
```

Sau khi đã biết cách biểu diễn dữ liệu, ta sẽ code các hàm cho `SkipLists`. Set "dỏm" của chúng ta sẽ gồm 6 hàm sau:

```
struct SkipLists {
    static const int MAX_LEVEL = 20;
    Column *head, *tail;

    SkipLists(); // Khởi tạo
    bool empty(); // Kiểm tra SkipLists có rỗng không
    Column *lower_bound(int); // Tìm vị trí Column chứa giá trị nhỏ nhất không nhỏ hơn giá trị cần tìm
    Column *upper_bound(int); // Tìm vị trí Column chứa giá trị nhỏ nhất lớn hơn giá trị cần tìm
    void insert(int); // Chèn một phần tử mang giá trị cho trước vào SkipLists
    void erase(int); // Xóa một phần tử mang giá trị cho trước khỏi SkipLists
};
```

Ta sẽ bắt đầu với constructor `SkipLists()`. Để khởi tạo `SkipLists`, ta sẽ tạo ra hai cột `head` và `tail` có chiều cao là số tầng tối đa, và tại liên kết giữa `head` và `tail` trên tất cả các `Cell`.

```
SkipLists::SkipLists() {
    head = new Column;
    tail = new Column;
    head->value = 0;
    tail->value = 0;
    for(int i = 0; i < MAX_LEVEL; i++) {
        head->cells.push_back((Cell){NULL, tail});
        tail->cells.push_back((Cell){head, NULL});
    }
}
```

Với hàm `empty()`, ta chỉ đơn giản kiểm tra liên kết cấp 0 (liên kết trực tiếp) của `head` có nối với `tail` không.

```
bool SkipLists::empty() {
    return head->cells[0].next_column == tail;
}
```

Với hàm `lower_bound()`, ta sẽ đi từ tầng cao nhất đến tầng thấp nhất, chừng nào nhảy về phía trước vẫn vào một phần tử có giá trị nhỏ hơn giá trị cần tìm thì ta cứ nhảy. Sau khi duyệt, ta sẽ đứng ở phần tử lớn nhất có giá trị nhỏ hơn giá trị cần tìm. Ta nhảy trên liên kết cấp 0 một lần nữa để lấy được `lower_bound()`.

```
Column *SkipLists::lower_bound(int value) {
    Column *iter = head;
    for(int level = MAX_LEVEL - 1; level >= 0; level--) {
        while(iter->cells[level].next_column != tail && iter->cells[level].next_column-
>value < value) {
            iter = iter->cells[level].next_column;
        }
    }
    return iter->cells[0].next_column;
}
```

Hàm `upper_bound()` không khác gì `lower_bound()`, ngoại trừ việc thay dấu < thành <= lúc so sánh với `value`.

```
Column *SkipLists::upper_bound(int value) {
    Column *iter = head;
    for(int level = MAX_LEVEL - 1; level >= 0; level--) {
        while(iter->cells[level].next_column != tail && iter->cells[level].next_column-
>value <= value) {
            iter = iter->cells[level].next_column;
        }
    }
    return iter->cells[0].next_column;
}
```

Với hàm `insert()`, ta sẽ chia thành 3 bước sau:

- Sử dụng `lower_bound()` để kiểm tra giá trị đã tồn tại trong `SkipLists` chưa. Nếu đã tồn tại, thoát khỏi hàm.
- Tạo ra một `Column` mới để chèn vào `SkipLists`. Ta sẽ sử dụng hàm `rand()` để tung đồng xu, xây dựng chiều cao cho `Column` này.

- Chèn `Column` vào `SkipLists`. Ta duyệt y như trong `lower_bound()` và `upper_bound()`, ở mỗi tầng chèn liên kết với `Column` vào hai cột đằng sau và đằng trước `Column`.

```
void SkipLists::insert(int value) {
    // Kiểm tra value đã tồn tại chưa
    Column *temp = lower_bound(value);
    if(temp != tail && temp->value == value) {
        return;
    }
    // Tạo inserted_column là cột chứa value để chèn vào SkipLists
    Column *inserted_column = new Column;
    inserted_column->value = value;
    inserted_column->cells.push_back((Cell){NULL, NULL});
    // Tung đồng xu tăng chiều cao
    while(inserted_column->cells.size() < MAX_LEVEL && rand() % 2 == 0) {
        inserted_column->cells.push_back((Cell){NULL, NULL});
    }
    // Duyệt để chèn
    Column *iter = head;
    for(int level = MAX_LEVEL - 1; level >= 0; level--) {
        while(iter->cells[level].next_column != tail && iter->cells[level].next_column-
>value < value) {
            iter = iter->cells[level].next_column;
        }
        if(level < inserted_column->cells.size()) {
            // Nối iter với inserted_column, nối inserted_column với next_iter
            Column *next_iter = iter->cells[level].next_column;
            iter->cells[level].next_column = inserted_column;
            next_iter->cells[level].previous_column = inserted_column;
            inserted_column->cells[level].previous_column = iter;
            inserted_column->cells[level].next_column = next_iter;
        }
    }
}
```

Với hàm `erase()`, ta sẽ chia thành 3 bước sau:

- Sử dụng `lower_bound()` để kiểm tra giá trị đã tồn tại trong `SkipLists` chưa. Nếu không tồn tại, thoát khỏi hàm.
- Xóa cột chứa giá trị cần xóa khỏi `SkipLists` bằng cách nối từng liên kết giữa các `Cell` liền trước và liền sau nó trên từng tầng.
- Xóa cột chứa giá trị cần xóa để giải phóng bộ nhớ.

```
void SkipLists::erase(int value) {
    // Kiểm tra value đã tồn tại chưa
    Column *erased_column = lower_bound(value);
    if(erased_column == tail || erased_column->value != value) {
        return;
    }
    // Duyệt để xóa
    Column *iter = head;
    for(int level = MAX_LEVEL - 1; level >= 0; level--) {
        while(iter->cells[level].next_column != tail && iter->cells[level].next_column-
>value <= value) {
            iter = iter->cells[level].next_column;
        }
        if(iter == erased_column) {
```

```
            // Nối previous_iter với next_iter
            Column *previous_iter = iter->cells[level].previous_column, *next_iter = iter-
>cells[level].next_column;
            previous_iter->cells[level].next_column = next_iter;
            next_iter->cells[level].previous_column = previous_iter;
        }
    }
    delete erased_column;
}
```

Với 6 hàm trên, bạn đã có thể mô phỏng một cách đơn giản một cái set "dỏm" để giải bài này. Bạn hãy thử tự làm tiếp và nộp trên SPOJ nhé. Toàn bộ code cho bài CPPSET có thể xem ở đây.

Mở rộng

- Ở trên mới là một code Skip Lists đơn giản nhất mô phỏng std::set để giải bài CPPSET. Liệu bạn có thể code lại một std::set hoàn hảo bằng Skip Lists không? Hãy thử xem!
- Code trên sử dụng cả liên kết xuôi (`next_column`) và liên kết ngược (`previous_column`) để dễ xử lí. Bạn có thể code lại CPPSET mà không cần sử dụng liên kết ngược không?
- Khi xây dựng cột để chèn vào Skip Lists, ta sử dụng kĩ thuật tung đồng xu với xác suất 1/2 mỗi mặt để xây dựng chiều cao cột. Tại sao phải là 1/2, liệu có thể là một con số khác không? Bạn hãy thử các con số khác nhau, sử dụng cả phân tích lý thuyết và thực nghiệm, cho thấy độ hiệu quả của các con số khác.
- Hẳn bạn sẽ thắc mắc dùng Skip Lists làm gì khi nó cũng chỉ để thay std::set, mà std::set thì có sẵn rồi. Skip Lists có rất nhiều ứng dụng và khả năng tùy biến nâng cao mà sẽ được giới thiệu trong phần 2 của bài viết này, giúp nó làm được những điều std::set không thể làm được, đơn giản nhất là tìm phần thử lớn thứ k trong tập hợp. Bạn thử tự nghĩ cách tìm phần tử lớn thứ k trong Skip Lists xem.

==========

****Red black tree**

Iterative-Tree-Search (x, k)

```
while x <> NIL and k <> key[x]
    do if k < key[x]
        then x := left[x]
        else x := right[x]
return x
```

Tree-Minimum(x)

```
while left[x] <> NIL
    do x := left[x]
return x
```

Tree-Maximum(x)

```
while right[x] <> NIL
    do x := right[x]
return x
```

Tree-Successor(x)

```
if right[x] <> NIL
    then return Tree-Minimum(right[x])
y := p[x]
while y <> NIL and x = right[y]
    do x := y
       y := p[y]
return y
```

Left-Rotate(T,x)

```
y := right[x]
right[x] := left[y]
if left[y] <> NIL
   then p[left[y]] := x
p[y] := p[x]
if p[x] = NIL
    then root[T] := y
    else if x = left[p[x]]
            then left[p[x]] := y
            else right[p[x]] := y
left[y] := x
p[x] := y
```

Right-Rotate(T,x)

this procedure is symmetric to the Left-rotate

RB-Insert(T,x)

```
Tree-Insert(T,x)
color[x] := RED
while x <> root[T] and color[p[x]] = RED
    do if p[x] = left[p[p[x]]]
          then y := right[p[p[x]]]
               if color[y] = RED
                  then color[p[x]] := BLACK
                       color[y] := BLACK
                       color[p[p[x]]] := RED
                       x := p[p[x]]
                  else if x = right[p[x]]
                          then x := p[x]
                               Left-Rotate(T,x)
                       color[p[x]] := BLACK
                       color[p[p[x]] := RED
                       Right-Rotate(T,p[p[x]])
          else (same as then clause
```

```
                with "right" and "left" exchanged)
color[root[T]] := BLACK
```

RB-Delete(*T,z*)

```
if left[z] = nil[T] or right[z] = nil[T]
   then y := z
   else y := Tree-Successor(z)
if left[y] <> nil[T]
   then x := left[y]
   else x := right[y]
p[x] := p[y]
if p[y] = nil[T]
   then root[T] := x
   else if y = left[p[y]]
           then left[p[y]] := x
           else right[p[y]] := x
if y <> z
   then key[z] := key[y]
        (if y has other fields, copy them, too)
if color[y] = BLACK
   then RB-Delete-Fixup(T,x)
return y
```

RB-Delete-Fixup(*T,x*)

```
while x <> root[T] and color[x] = BLACK
    do if x = left[p[x]]
          then w := right[p[x]]
               if color[w] = RED
                  then color[w] := BLACK
                       color[p[x]] := RED
                       Left-Rotate(T,p[x])
                       w := right[p[x]]
                if color[left[w]] = BLACK and color[right[w]] = BLACK
                   then color[w] := RED
                        x := p[x]
                   else if color[right[w]] = BLACK
                           then color[left[w]] := BLACK
                                color[w] := RED
                                Right-Rotate(T,w)
                                w := right[p[x]]
                        color[w] := color[p[x]]
                        color[p[x]] := BLACK
                        color[right[w]] := BLACK
                        Left-Rotate(T,p[x])
                        x := root[T]
       else (same as then clause
             with "right" and "left" exchanged)
color[x] := BLACK
```

==========

****Btree**

```
// C++ implemntation of search() and traverse() methods
#include<iostream>
```

```
using namespace std;

// A BTree node
class BTreeNode
{
    int *keys;  // An array of keys
    int t;      // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n;     // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf);   // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k);   // returns NULL if k is not present.

// Make BTree friend of this so that we can access private members of this
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t;  // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {  root = NULL;  t = _t; }

    // function to traverse the tree
    void traverse()
    {  if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    {  return (root == NULL)? NULL : root->search(k); }
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int _t, bool _leaf)
{
    // Copy the given minimum degree and leaf property
    t = _t;
    leaf = _leaf;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}
```

```
// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}
```

***insert

```
// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
        root->keys[0] = k;  // Insert key
        root->n = 1;  // Update number of keys in root
```

```
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {
            // Allocate memory for new root
            BTreeNode *s = new BTreeNode(t, false);

            // Make old root as child of new root
            s->C[0] = root;

            // Split the old root and move 1 key to the new root
            s->splitChild(0, root);

            // New root has two children now.  Decide which of the
            // two children is going to have new key
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);

            // Change root
            root = s;
        }
        else  // If root is not full, call insertNonFull for root
            root->insertNonFull(k);
    }
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;
```

```
    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two.  See which of the two
            // is going to have the new key
            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insertNonFull(k);
    }
}
```

```
// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreeNode::splitChild(int i, BTreeNode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];
```

```
    // Increment count of keys in this node
    n = n + 1;
}
```

***delete

```
// A utility function that returns the index of the first key that is
// greater than or equal to k
int BTreeNode::findKey(int k)
{
    int idx=0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}

// A function to remove the key k from the sub-tree rooted with this node
void BTreeNode::remove(int k)
{
    int idx = findKey(k);

    // The key to be removed is present in this node
    if (idx < n && keys[idx] == k)
    {

        // If the node is a leaf node - removeFromLeaf is called
        // Otherwise, removeFromNonLeaf function is called
        if (leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    }
    else
    {

        // If this node is a leaf node, then the key is not present in tree
        if (leaf)
        {
            cout << "The key "<< k <<" is does not exist in the tree\n";
            return;
        }

        // The key to be removed is present in the sub-tree rooted with this node
        // The flag indicates whether the key is present in the sub-tree rooted
        // with the last child of this node
        bool flag = ( (idx==n)? true : false );

        // If the child where the key is supposed to exist has less that t keys,
        // we fill that child
        if (C[idx]->n < t)
            fill(idx);

        // If the last child has been merged, it must have merged with the
previous
        // child and so we recurse on the (idx-1)th child. Else, we recurse on the
```

```
        // (idx)th child which now has atleast t keys
        if (flag && idx > n)
            C[idx-1]->remove(k);
        else
            C[idx]->remove(k);
    }
    return;
}

// A function to remove the idx-th key from this node - which is a leaf node
void BTreeNode::removeFromLeaf (int idx)
{

    // Move all the keys after the idx-th pos one place backward
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Reduce the count of keys
    n--;

    return;
}

// A function to remove the idx-th key from this node - which is a non-leaf node
void BTreeNode::removeFromNonLeaf(int idx)
{

    int k = keys[idx];

    // If the child that precedes k (C[idx]) has atleast t keys,
    // find the predecessor 'pred' of k in the subtree rooted at
    // C[idx]. Replace k by pred. Recursively delete pred
    // in C[idx]
    if (C[idx]->n >= t)
    {
        int pred = getPred(idx);
        keys[idx] = pred;
        C[idx]->remove(pred);
    }

    // If the child C[idx] has less that t keys, examine C[idx+1].
    // If C[idx+1] has atleast t keys, find the successor 'succ' of k in
    // the subtree rooted at C[idx+1]
    // Replace k by succ
    // Recursively delete succ in C[idx+1]
    else if  (C[idx+1]->n >= t)
    {
        int succ = getSucc(idx);
        keys[idx] = succ;
        C[idx+1]->remove(succ);
    }

    // If both C[idx] and C[idx+1] has less that t keys,merge k and all of
C[idx+1]
    // into C[idx]
    // Now C[idx] contains 2t-1 keys
    // Free C[idx+1] and recursively delete k from C[idx]
    else
    {
```

```
        merge(idx);
        C[idx]->remove(k);
    }
    return;
}

// A function to get predecessor of keys[idx]
int BTreeNode::getPred(int idx)
{
    // Keep moving to the right most node until we reach a leaf
    BTreeNode *cur=C[idx];
    while (!cur->leaf)
        cur = cur->C[cur->n];

    // Return the last key of the leaf
    return cur->keys[cur->n-1];
}

int BTreeNode::getSucc(int idx)
{

    // Keep moving the left most node starting from C[idx+1] until we reach a leaf
    BTreeNode *cur = C[idx+1];
    while (!cur->leaf)
        cur = cur->C[0];

    // Return the first key of the leaf
    return cur->keys[0];
}

// A function to fill child C[idx] which has less than t-1 keys
void BTreeNode::fill(int idx)
{

    // If the previous child(C[idx-1]) has more than t-1 keys, borrow a key
    // from that child
    if (idx!=0 && C[idx-1]->n>=t)
        borrowFromPrev(idx);

    // If the next child(C[idx+1]) has more than t-1 keys, borrow a key
    // from that child
    else if (idx!=n && C[idx+1]->n>=t)
        borrowFromNext(idx);

    // Merge C[idx] with its sibling
    // If C[idx] is the last child, merge it with with its previous sibling
    // Otherwise merge it with its next sibling
    else
    {
        if (idx != n)
            merge(idx);
        else
            merge(idx-1);
    }
    return;
}

// A function to borrow a key from C[idx-1] and insert it
// into C[idx]
```

```
void BTreeNode::borrowFromPrev(int idx)
{

    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx-1];

    // The last key from C[idx-1] goes up to the parent and key[idx-1]
    // from parent is inserted as the first key in C[idx]. Thus, the  loses
    // sibling one key and child gains one key

    // Moving all key in C[idx] one step ahead
    for (int i=child->n-1; i>=0; --i)
        child->keys[i+1] = child->keys[i];

    // If C[idx] is not a leaf, move all its child pointers one step ahead
    if (!child->leaf)
    {
        for(int i=child->n; i>=0; --i)
            child->C[i+1] = child->C[i];
    }

    // Setting child's first key equal to keys[idx-1] from the current node
    child->keys[0] = keys[idx-1];

    // Moving sibling's last child as C[idx]'s first child
    if (!leaf)
        child->C[0] = sibling->C[sibling->n];

    // Moving the key from the sibling to the parent
    // This reduces the number of keys in the sibling
    keys[idx-1] = sibling->keys[sibling->n-1];

    child->n += 1;
    sibling->n -= 1;

    return;
}

// A function to borrow a key from the C[idx+1] and place
// it in C[idx]
void BTreeNode::borrowFromNext(int idx)
{

    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx+1];

    // keys[idx] is inserted as the last key in C[idx]
    child->keys[(child->n)] = keys[idx];

    // Sibling's first child is inserted as the last child
    // into C[idx]
    if (!(child->leaf))
        child->C[(child->n)+1] = sibling->C[0];

    //The first key from sibling is inserted into keys[idx]
    keys[idx] = sibling->keys[0];

    // Moving all keys in sibling one step behind
    for (int i=1; i<sibling->n; ++i)
```

```
        sibling->keys[i-1] = sibling->keys[i];

    // Moving the child pointers one step behind
    if (!sibling->leaf)
    {
        for(int i=1; i<=sibling->n; ++i)
            sibling->C[i-1] = sibling->C[i];
    }

    // Increasing and decreasing the key count of C[idx] and C[idx+1]
    // respectively
    child->n += 1;
    sibling->n -= 1;

    return;
}

// A function to merge C[idx] with C[idx+1]
// C[idx+1] is freed after merging
void BTreeNode::merge(int idx)
{
    BTreeNode *child = C[idx];
    BTreeNode *sibling = C[idx+1];

    // Pulling a key from the current node and inserting it into (t-1)th
    // position of C[idx]
    child->keys[t-1] = keys[idx];

    // Copying the keys from C[idx+1] to C[idx] at the end
    for (int i=0; i<sibling->n; ++i)
        child->keys[i+t] = sibling->keys[i];

    // Copying the child pointers from C[idx+1] to C[idx]
    if (!child->leaf)
    {
        for(int i=0; i<=sibling->n; ++i)
            child->C[i+t] = sibling->C[i];
    }

    // Moving all keys after idx in the current node one step before -
    // to fill the gap created by moving keys[idx] to C[idx]
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Moving the child pointers after (idx+1) in the current node one
    // step before
    for (int i=idx+2; i<=n; ++i)
        C[i-1] = C[i];

    // Updating the key count of child and the current node
    child->n += sibling->n+1;
    n--;

    // Freeing the memory occupied by sibling
    delete(sibling);
    return;
}
```

```
// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

void BTree::remove(int k)
{
    if (!root)
    {
        cout << "The tree is empty\n";
        return;
    }

    // Call the remove function for root
    root->remove(k);

    // If the root node has 0 keys, make its first child as the new root
    //  if it has a child, otherwise set root as NULL
    if (root->n==0)
    {
        BTreeNode *tmp = root;
        if (root->leaf)
```

```
            root = NULL;
        else
            root = root->C[0];

        // Free the old root
        delete tmp;
    }
    return;
}
```

Yêu cầu đối với hàm băm tốt:

Một hàm băm tốt thường phải thỏa các yêu cầu sau:

- Phải giảm thiểu sự xung đột.
- Phải phân bố đều các phần tử trên M địa chỉ khác nhau của bảng băm.

4. CÁC CÁCH GIẢI QUYẾT XUNG ĐỘT

Như đã đề cập ở phần trên, sự xung đột là hiện tượng các khóa khác nhau nhưng băm cùng địa chỉ như nhau, hay ánh xạ vào cùng một địa chỉ

Một cách tổng quát, khi key1<>key2 mà f(key1)=f(key2) chúng ta nói phần tử có khóa key1 xung đột với phần tử có khóa key2.

Thực tế người ta giải quyết sự xung đột theo hai phương pháp: ***phương pháp nối kết*** và ***phương pháp băm lại***.

Giải quyết sự xung đột bằng phương pháp nối kết:

Các phần tử bị băm cùng địa chỉ (các phần tử bị xung đột) được gom thành một danh sách liên kết. Lúc này mỗi phần tử trên bảng băm cần khai báo thêm trường liên kết next chỉ phần tử kế bị xung đột cùng địa chỉ.

Bảng băm giải quyết sự xung đột bằng phương pháp này cho phép tổ chức các phần tử trên bảng băm rất linh hoạt: khi thêm một phần tử vào bảng băm chúng ta sẽ thêm phần tử này vào danh sách liên kết thích hợp phụ thuộc vào băm. Tuy nhiên bảng bảng băm loại này bị hạn chế về tốc độ truy xuất.

Các loại bảng băm giải quyết sự xung đột bằng phương pháp nối kết như: bảng băm với phương pháp nối kết trực tiếp, bảng băm với phương pháp nối kết hợp nhất.

Giải quyết sự xung đột bằng phương pháp băm lại:

Nếu băm lần đầu bị xung đột thì băm lại lần 1, nếu bị xung đột nữa thì băm lai lần 2,… Quá trình băm lại diễn ra cho đến khi không còn xung đột nữa. Các phép băm lại (rehash function) thường sẽ chọn địa chỉ khác cho các phần tử.

Để tăng tốc độ truy xuất, các bảng băm giải quyết sự xung đột bằng phương pháp băm lại thường được cài đặt bằng danh sách kề. Tuy nhiên việc tổ chức các phần tử trên bảng băm không linh hoạt vì các phần tử chỉ được lưu trữ trên một danh sách kề có kích thước đã xác định trước.

Các loại bảng băm giải quyết sự xung đột bằng phương pháp băm lại như: bảng băm với phương pháp dò tuyến tính, bảng băm với phương pháp dò bậc hai, bảng băm với phương pháp băm kép.

4.1. Bảng băm với phương pháp nối kết trực tiếp (Direct chaining Method)

Mô tả: Xem hình vẽ

Hình 1.6. bảng băm với phương pháp nối kết trực tiếp

Bảng băm được cài đặt bằng các danh sách liên kết, các phần tử trên bảng băm được "băm" thành M danh sách liên kết (từ danh sách 0 đến danh sách M–1). Các phần tử bị xung đột tại địa chỉ i được nối kết trực tiếp với nhau qua danh sách liên kết i. Chẳng hạn, với M=10, các phần tử có hàng đơn vị là 9 sẽ được băm vào danh sách liên kết i = 9.

Khi thêm một phần tử có khóa k vào bảng băm, hàm băm f(k) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách liên kết i mà phần tử này sẽ được thêm vào.

Khi tìm một phần tử có khóa k vào bảng băm, hàm băm f(k) cũng sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách liên kết i có thể chứa phần tử này. Như vậy, việc tìm kiếm phần tử trên bảng băm sẽ được qui về bài toán tìm kiếm một phần tử trên danh sách liên kết.

Để minh họa cho vấn đề vừa nêu:

Xét bảng băm có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}
- Hàm băm f(key) = key % 10.

Hình trên minh họa bảng băm vừa mô tả. Theo hình vẽ, bảng băm đã "*băm*" phần tử trong tập khoá K theo 10 danh sách liên kết khác nhau, mỗi danh sách liên kết gọi là một bucket:

- Bucket 0 gồm những phần tử có khóa tận cùng bằng 0.
- Bucket i(i=0 | … | 9) gồm những phần tử có khóa tận cùng bằng i. Để giúp việc truy xuất bảng băm dễ dàng, các phần tử trên các bucket cần thiết được tổ chức theo một thứ tự, chẳng hạn từ nhỏ đến lớn theo khóa.
- Khi khởi động bảng băm, con trỏ đầu của các bucket là NULL.

Theo cấu trúc này, với tác vụ insert, hàm băm sẽ được dùng để tính địa chỉ của khoá k của phần tử cần chèn, tức là xác định được bucket chứa phần tử và đặt phần tử cần chèn vào bucket này.

Với tác vụ search, hàm băm sẽ được dùng để tính địa chỉ và tìm phần tử trên bucket tương ứng.

Cài đặt bảng băm dùng phương pháp nối kết trực tiếp :

a. Khai báo cấu trúc bảng băm:

```
#define     M     100
typedef struct tagNODE
{
    int          key;
    tagNODE *next
}NODE, *NODEPTR;

/* khai bao mang bucket chua M con tro dau cua Mbucket */
NODEPTR bucket[M];
```

b.Các phép toán:

Hàm băm

Giả sử chúng ta chọn hàm băm dạng %: f(key)=key % M.

```
int hashfunc (int key)
{
    return (key % M);
}
```

Chúng ta có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

Phép toán initbuckets:

Khởi động các bucket.

```
void initbuckets( )
{
    for (int b=0;  b<M;  b++);
    bucket[b] = NULL;
}
```

Phép toán emmptybucket:

Kiểm tra bucket b có bị rỗng không?

```
int emptybucket (int b)
{
    return (bucket[b] ==NULL ?TRUE :FALSE);
}
```

Phép toán emmpty:

Kiểm tra bảng băm có rỗng không?

```
int empty( )
{
    int b;
    for (b = 0;b<M;b++)
    if(bucket[b] !=NULL)  return(FALSE);
    return(TRUE);
}
```

Phép toán insert:

Thêm phần tử có khóa k vào bảng băm.

Giả sử các phần tử trên các bucket là có thứ tự để thêm một phần tử khóa k vào bảng băm trước tiên chúng ta xác định bucket phù hợp, sau đó dùng phép toán place của danh sách liên kết để đặt phần tử vào vi trí phù hợp trên bucket.

```
void insert(int k)
{
    int b;
    b= hashfunc(k)
    place(b,k); //tac vu place cua danh sach lien ket
}
```

Phép toán remove:

Xóa phần tử có khóa k trong bảng băm.

Giả sử các phần tử trên các bucket là có thứ tự, để xóa một phần tử khóa k trong bảng băm cần thực hiện:

- Xác định bucket phù hợp
- Tìm phần tử để xóa trong bucket đã được xác định, nếu tìm thấy phần tử cần xóa thì loại bỏ phần tử theo các phép toán tương tự loại bỏ một phần tử trong danh sách liên kết.

```
void remove ( int k)
{
        int b;
        NODEPTR q, p;
        b = hashfunc(k);
        p = hashbucket(k);
        q=p;
        while(p !=NULL && p->key !=k)
        {
                q=p;
                p=p->next;
        }
        if (p == NULL)
                printf("\n khong co nut co khoa %d" ,k);
        else
                if (p == bucket [b])   pop(b);
                //Tac vu pop cua danh sach lien ket
                else
                        delafter(q);
        /*tac vu delafter cua danh sach lien ket*/
}
```

Phép toán clearbucket:

Xóa tất cả các phần tử trong bucket b.

```
void clearbucket (int b)
{
        NODEPTR p,q;
        //q la nut truoc,p la nut sau
        q = NULL;
        p = bucket[b];
        while(p !=NULL)
        {
                q = p;
                p=p->next;
                freenode(q);
        }
        bucket[b] = NULL; //khoi dong lai butket b
}
```

Phép toán clear:

Xóa tất cả các phần tử trong bảng băm.

```
void clear( )
{
        int b;
        for (b = 0; b<M ; b++)
        clearbucket(b);
}
```

Phép toán traversebucket:

Duyệt các phần tử trong bucket b.

```
void traversebucket (int b)
{
        NODEPTR p;
        p= bucket[b];
        while (p !=NULL)
        {
                printf("%3d", p->key);
```

```
                p= p->next;
            }
        }
```

Phép toán traverse:

Duyệt toàn bộ bảng băm.

```
void traverse( )
{
        int b;
        for (b = 0;n<M; b++)
        {
                printf("\nButket %d:",b);
                traversebucket(b);
        }
}
```

Phép toán search:

Tìm kiếm một phần tử trong bảng băm,nếu không tìm thấy hàm này trả về hàm NULL,nếu tìm thấy hàm này trả về con trả chỉ tìm phần tử tìm thấy.

```
NODEPTR search(int k)
{
        NODEPTR p;
        int b;
        b = hashfunc (k);
        p = bucket[b];
        while(k > p->key && p !=NULL)
        p=p->next;
        if (p == NULL | | k !=p->key)// khong tim thay
                return(NULL);
                else//tim thay
        // else //tim thay
        return(p);
}
```

Nhận xét bảng băm dùng phương pháp nối kết trực tiếp :

Bảng băm dùng phương pháp nối kết trực tiếp sẽ "băm" n phần tử vào danh sách liên kết (M bucket).

Để tốc độ thực hiện các phép toán trên bảng hiệu quả thì cần chọn hàm băm sao cho băm đều n phần tử của bảng băm cho M bucket, lúc này trung bình mỗi bucket sẽ có n/M phần tử. Chẳng hạn, phép toán *search* sẽ thực hiện việc tìm kiếm tuyến tính trên bucket nên thời gian tìm kiếm lúc này có bậc 0 (n/M) – nghĩa là, nhanh gấp n lần so với việc tìm kiếm trên một danh sách liên kết có n phần tử.

Nếu chọn M càng lớn thì tốc độ thực hiện các phép toán trên bảng băm càng nhanh, tuy nhiên lại càng dùng nhiều bộ nhớ. Do vậy, cần điều chỉnh M để dung hòa giữa tốc độ truy xuất và dung lượng bộ nhớ.

- Nếu chọn M=n thì năng xuất tương đương với truy xất trên mảng (có bậc O(1)), tuy nhiên tốn nhiều bộ nhớ.
- Nếu chọn M =n /k(k =2,3,4,…) thì ít tốn bộ nhớ hơn k lần, nhưng tốc độ chậm đi k lần.

Chương trình minh họa: http://sites.google.com/site/ngo2uochung/courses/hashtable-directchaining

4.2. Bảng băm với phương pháp nối kết hợp nhất (Coalesced chaining Method)

Mô tả:

- *Cấu trúc dữ liệu:* Tương tự như trong trường hợp cài đặt bằng phương pháp nối kết trực tiếp, bảng băm trong trường hợp này được cài đặt bằng danh sách liên kết dùng mảng, có M phần tử. Các phần tử bị xung đột tại một địa chỉ được nối kết nhau qua một danh sách liên kết. Mỗi phần tử của bảng băm gồm hai trường:
  - Trường key: chứa khóa của mỗi phần tử
  - Trường next: con trỏ chỉ đến phần tử kế tiếp nếu có xung đột.
- *Khởi động*: Khi khởi động, tất cả trường key của các phần tử trong bảng băm được gán bởi giá trị Null, còn tất cả các trường next được gán –1.

- *Thêm mới một phần tử:* Khi thêm mới một phần tử có khóa key vào bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1.
  - Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.
  - Nếu bị xung đột thì phần tử mới được cấp phát là phần tử trống phía cuối mảng. Cập nhật liên kết next sao cho các phần tử bị xung đột hình thành một danh sách liên kết.
- *Tìm kiếm*: Khi tìm kiếm một phần tử có khóa key trong bảng băm, hàm băm f(key) sẽ giúp giới hạn phạm vi tìm kiếm bằng cách xác định địa chỉ i trong khoảng từ 0 đến M-1, và việc tìm kiếm phần tử khóa có khoá key trong danh sách liên kết sẽ xuất phát từ địa chỉ i.

Để minh họa cho bảng băm với phương pháp nối kết hợp nhất, xét ví dụ sau:
Giả sử, khảo sát bảng băm có cấu trúc như sau:
- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}
- Hàm băm f(key) = key % 10.

Key : A C B D E
Hash: 1 2 1 1 3

key	next
NULL	-1
NULL	-1
...	...
NULL	-1

0	NULL	-1
1	A	M-1
2	C	-1
3	E	-1
...	...	
M-2	D	-1
M-1	B	M-2

Cài đặt bảng băm dùng phương pháp nối kết hợp nhất:

a. Khai báo cấu trúc bảng băm:

```
#define NULLKEY –1
#define M 100
/* M la so nut co tren bang bam, du de chua cac nut nhap vao bang bam */
// Khai bao cau truc mot nut cua bang bam
struct node
{
        int key; //khoa cua nut tren bang bam
        int next;
        //con tro chi nut ke tiep khi co xung dot
};

//Khai bao bang bam
struct node hashtable[M];
int avail;
/* bien toan cuc chi nut trong o cuoi table duoc cap nhat khi co xung dot */
```

b. Các tác vụ:

Hàm băm:

Giả sử chúng ta chọn hàm băm dạng modulo: f(key)=key % 10.

```
int hashfunc(int key)
{
        return(key % 10);
}
```

Chúng ta có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

Phép toán khởi tạo (Initialize):

Phép toán này cho khởi động bảng băm: gán tất cả các phần tử trên bảng có trường key là Null, trường next là –1.

Gán biến toàn cục avail=M-1, là phần tử cuối danh sách chuẩn bị cấp phát néu xãy ra xung đột.

```
void initialize()
{
	for(int i = 0;i<M;i++)
	{
		hashtable[i].key = NULLKEY;
		hashtable[i].key = -1;
	}
	avail = M-1;
	/* nut M-1 la nut o cuoi bang chuan bi cap phat neu co xung dot*/
}
```

Phép toán kiểm tra rỗng (empty):

Kiểm tra bảng băm có rỗng không.

```
int empty ();
{
	int i;
	for(i = 0;i< M;i++)
	if(hashtable[i].key !=NULLKEY)
		return(FALSE);
	return(TRUE);
}
```

Phép toán tìm kiếm (search):

Tìm kiếm theo phương pháp tuyến tính, nếu không tìm thấy hàm tìm kiếm trả về trị M, nếu tìm thấy hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
	int i;
	i=hashfunc(k);
	while(k !=hashtable[i].key && i !=-1)
		i=hashtable[i].next;
	if(k== hashtable[i]key)
		return(i);//tim thay
	return(M);//khong tim thay
}
```

Phép toán lấy phần tử trống (Getempty):

Chọn phần tử còn trống phía cuối bản băm để cấp phát khi xảy ra xung đột.

```
int getempty()
{
	while(hashtable[avail].key !=NULLKEY)  avail - -;
	return(avail);
}
```

Phép toán chèn phần tử mới vào bảng băm (insert):

Thêm phần tử có khóa k vào bảng băm.

```
int insert(int k)
{
	int i;
	//con tro lan theo danh sach lien ket chua cac nut //bi xung dot
	int j;
	//dia chi nut trong duoc cap phat
	i = search(k);
	if(i !=M)
	{
		printf("\n khoa %d bi trung,khong them nut nay duoc",k);
		return(i);
	}
	i=hashfunc(k);
	while(hashtable[i]next >=0) i=hashtable[i].next;
	if(hashtable[i].key == NULLKEY)
		//Nut i con trong thi cap nhat
```

```
                j = i;
        else
        //Neu nut i la nut cuoi cua DSLK
        {
                j = getempty();
                if(j < 0)
                {
                        printf("\n Bang bam bi day,khongthem nut co khoa %d
                        duoc"k);
                        return(j);
                }
                else
                        hashtable[i].next = j;
        }
        hashtable[j].key = k;
        return(j);
}
```

Nhận xét bảng băm dùng phương pháp nối kết hợp nhất:

Thực chất cấu trúc bảng băm này chỉ tối ưu khi băm đều, nghĩa là mỗi danh sách liên kết chứa một vài phần tử bị xung đột, tốc độ truy xuất lúc này có bậc 0(1). Trường hợp xấu nhất là băm không đều vì hình thành một danh sách có n phần tử nên tốc độ truy xuất lúc này có bậc 0(n).

Chương trình minh họa:

Chương trình Hashtable, dùng phương pháp nối kết hợp nhất *(coalesced chaining method)* - Cài đặt bằng danh sách kề.

http://sites.google.com/site/ngo2uochung/courses/hashtable-coalescedchaining

4.3. Bảng băm với phương pháp dò tuyến tính (Linear Probing Method)

Mô tả:

- *Cấu trúc dữ liệu:* Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm là một mẫu tin có một trường key để chứa khoá của phần tử.

Khi khởi động bảng băm thì tất cả trường key được gán Null

- *Khi thêm phần tử* có khoá key vào bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1:
  - Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.
  - Nếu bị xung đột thì hàm băm lại lần 1, hàm f_1 sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm thì hàm băm lại lần 2, hàm f_2 sẽ xét địa chỉ kế tiếp nữa, …, và quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử mới vào địa chỉ này.
- *Khi tìm một phần tử* có khoá key trong bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1, tìm phần tử khoá key trong khối đặt chứa các phần tử xuất phát từ địa chỉ i.

Hàm băm lại của phương pháp dò tuyến tính là truy xuất địa chỉ kế tiếp. Hàm băm lại lần i được biểu diễn bằng công thức sau:

f(key)=(f(key)+i) %M với f(key) là hàm băm chính của bảng băm.

Lưu ý địa chỉ dò tìm kế tiếp là địa chỉ 0 nếu đã dò đến cuối bảng.

Giả sử, khảo sát bảng băm có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}
- Hàm băm f(key) = key % 10.

Hình thể hiện thêm các nut 32, 53, 22, 92, 17, 34, 24, 37, 56 vào bảng băm.

0	NULL	0	NULL	0	NULL	0	NULL	0	56
1	NULL	1	NULL	1	NULL	1	NULL	1	NULL
2	32	2	32	2	32	2	32	2	32
3	53	3	53	3	53	3	53	3	53

4	NULL	4	22	4	22	4	22	4	22
5	NULL	5	92	5	92	5	92	5	92
6	NULL	6	NULL	6	34	6	34	6	34
7	NULL	7	NULL	7	17	7	17	7	17
8	NULL	8	NULL	8	NULL	8	24	8	24
9	NULL	9	NULL	9	NULL	9	37	9	37

Cài đặt bảng băm dùng phương pháp dò tuyến tính:

a. Khai báo cấu trúc bảng băm:

```
#define NULLKEY –1
#define M 100
/*
M la so nut co tren bang bam,du de chua cac nut nhap vao bang bam
*/
//khai bao cau truc mot nnut cua bang bam
struct node
{
        int key; //khoa cua nut tren bang bam
};

//Khai bao bang bam co M nut
struct node hashtable[M];
int NODEPTR;
/* bien toan cuc chi so nut hien co tren bang bam */
```

b. Các tác vụ:

Hàm băm:

Giả sử chúng ta chọn hàm băm dạng%:f(key0=key %10.

```
int hashfunc(int key)
{
        return(key% 10);
}
```

Chúng ta có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

Phép toán khởi tạo (initialize):

Khởi tạo bảng băm.

Gán tất cả các phần tử trên bảng có trường key là NULL.

Gán biến toàn cục N=0.

```
void initialize( )
{
        int i;
        for(i=0;i<M;i++)
                hashtable[i].key=NULLKEY;
        N=0;
        //so nut hien co khoi dong bang 0
}
```

Phép toán kiểm tra trống (empty):

Kiểm tra bảng băm có trống hay không.

```
int empty( );
{
        return(N==0 ? TRUE;FALSE);
}
```

Phép toán kiểm tra đầy (full):

Kiểm tra bảng băm đã đầy chưa.

```
int full( )
{
        return (N==M-1 ? TRUE; FALSE);
}
```

Lưu ý bảng băm đầy khi N=M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.

Phép toán search:

Việc tìm kiếm phần tử có khoá k trên một khối đặc, bắt đầu từ một địa chỉ i = HF(k), nếu không tìm thấy phần tử có khoá k, hàm này sẽ trả về trị M, còn nếu tìm thấy, hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
        int i;
        i=hashfunc(k);
        while(hashtable[i].key!=k && hashtable[i].key !=NULKEY)
        {
                //bam lai (theo phuong phap do tuyen tinh:fi(key)=f(key)+) % M
                i=i+1;
                if(i>=M)
                i=i-M;
        }
        if(hashtable[i].key==k) //tim thay
                return(i);
        else
        //khong tim thay
        return(M);
}
```

Phép toán insert:

Thêm phần tử có khoá k vào bảng băm.

```
int insert(int k)
{
        int i, j;
        if(full( ))
        {
                printf("\n Bang bam bi day khong them nut co khoa %d duoc",k);
                return;
        }
        i=hashfunc(k);
        while(hashtable[i].key !=NULLKEY)
        {
                //Bam lai (theo phuong phap do tuyen tinh)
                i ++;
                if(i >M)  i= i-M;
        }
        hashtable[i].key=k;
        N=N+1;
        return(i);
}
```

Nhận xét bảng băm dùng phương pháp dò tuyến tính:

Bảng băm này chỉ tối ưu khi băm đều, nghĩa là, trên bảng băm các khối đặc chứa vài phần tử và các khối phần tử chưa sử dụng xen kẻ nhau, tốc độ truy xuất lúc này có bậc 0(1). Trường hợp xấu nhất là băm không đều hoặc bảng băm đầy, lúc này hình thành một khối đặc có n phần tử, nên tốc độ truy xuất lúc này có bậc 0(n).

Chương trình minh họa:

Bảng băm, dùng phương pháp dò tuyến tính (linear proping method)-cài đặt bằng danh sách kề.

http://sites.google.com/site/ngo2uochung/courses/hashtable-linearproping

4.4. Bảng băm với phương pháp dò bậc hai (Quadratic Probing Method)

Mô tả:

- *Cấu trúc dữ liệu:* Bảng băm dùng phương pháp dò tuyến tính bị hạn chế do rải các phần tử không đều, bảng băm với phương pháp dò bậc hai rải các phần tử đều hơn.
Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm là một mẫu tin có một trường key để chứa khóa các phần tử.
- *Khi khởi động* bảng băm thì tất cả trường key bị gán NULL.
Khi thêm phần tử có khóa key vào bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1.

- Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ i này.
- Nếu bị xung đột thì hàm băm lại lần 1 f1 sẽ xác định địa chỉ cách 1^2, nếu lại bị xung đột thì hàm băm lại lần 2 f2 sẽ xét địa chỉ cách i 2^2 ,… , quá trình cứ thế cho đến khi nào tìm được trống và thêm phần tử vào địa chỉ này.

- *Khi tìm kiếm* một phần tử có khóa key trong bảng băm thì xét phần tử tại địa chỉ i=f(key), nếu chưa tìm thấy thì xét phần tử cách i 1^2, 2^2, ..., quá trình cứ thế cho đến khi tìm được khóa (trường hợp tìm thấy) hoặc rơi vào địa chỉ trống (trường hợp không tìm thấy).
- *Hàm băm* lại của phương pháp dò bậc hai là truy xuất các địa chỉ cách bậc 2. Hàm băm lại hàm i được biểu diễn bằng công thức sau:

$$fi(key)=(f(key) + i^2) \% M$$

với f(key) là hàm băm chính của bảng băm.
Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.
Bảng băm với phương pháp do bậc hai nên chọn số địa chỉ M là số nguyên tố.
Bảng băm minh họa có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}
- Hàm băm f(key) = key % 10.

Cài đặt bảng băm dùng phương pháp dò bậc hai:

a. Khai báo cấu trúc bảng băm:

```
#define NULLKEY –1
#define M 101
/*
M la so nut co tren bang bam,du de chua cac nut
nhap vao bang bam,chon M la so nguyen to
*/
//Khai bao nut cua bang bam
struct node
{
        int key; //Khoa cua nut tren bang bam
};
//Khai bao bang bam co M nut
struct node hashtable[M];
int N;
//Bien toan cuc chi so nut hien co tren bang bam
```

b. Các tác vụ :

Hàm băm:

Giả sử chúng ta chọn hàm băm dạng%: f(key)=key %10.

```
int hashfunc(int key)
{
        return(key% 10);
}
```

Chúng ta có thể dùng một hàm băm bất kì tahy cho hàm băm dạng % trên.

Phép toán initialize

Khởi động hàm băm.
Gán tất cả các phần tử trên bảng có trường key là NULLKEY.
Gán biến toàn cục N=0.

```
void initialize()
```

```
{
        int i;
        for(i=0; i<M;i++)  hashtable[i].key = NULLKEY;
        N=0; //so nut hien co khoi dong bang 0
}
```

Phép toán empty:

Kiểm tra bảng băm có rỗng không

```
int empty()
{
    return(N ==0 ?TRUE :FALSE);
}
```

Phép toán full:

Kiểm tra bảng băm đã đầy chưa .

```
int full()
{
        return(N = = M-1 ?TRUE :FALSE);
}
```

Lưu ý bảng băm đầy khi N=M-1 chúng ta nên chừa ít nhất một phần tử trong trên bảng băm!

Phép toán search:

Tìm phần tử có khóa k trên bảng băm,nếu không tìm thấy hàm này trả về trị M, nếu tìm thấy hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
        int i, d;
        i = hashfuns(k);
        d = 1;
        while(hashtable[i].key!=k&&hashtable[i].key !=NULLKEY)
        {
                //Bam lai (theo phuong phap bac hai)
                i = (i+d) % M;
                d = d+2;
        }
        hashtable[i].key =k;
        N = N+1;
        return(i);
}
```

Nhận xét bảng băm dùng phương pháp dò bậc hai:

Nên chọn số địa chỉ M là số nguyên tố. Khi khởi động bảng băm thì tất cả M trường key được gán NULL, biến toàn cục N được gán 0.

Bảng băm đầy khi N = M-1, và nên dành ít nhất một phần tử trống trên bảng băm.

Bảng băm này tối ưu hơn bảng băm dùng phương pháp dò tuyến tính do rải rác phần tử đều hơn, nếu bảng băm chưa đầy thì tốc độ truy xuất có bậc 0(1). Trường hợp xấu nhất là bảng băm đầy vì lúc đó tốc độ truy xuất chậm do phải thực hiện nhiều lần so sánh.

2.4.5. Bảng băm với phương pháp băm kép (Double hashing Method)

Mô tả:

- *Cấu trúc dữ liệu:* Bảng băm này dùng hai hàm băm khác nhau với mục đích để rải rác đều các phần tử trên bảng băm.

Chúng ta có thể dùng hai hàm băm bất kì, ví dụ chọn hai hàm băm như sau:

f1(key)= key %M.

f2(key) =(M-2)-key %(M-2).

bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm là một mẫu tin có một trường key để lưu khoá các phần tử.

- *Khi khởi động* bảng băm,tất cả trường kay được gán NULL.
- *Khi thêm phần tử* có khoá key vào bảng băm, thì i=f1(key) và j=f2(key) sẽ xác định địa chỉ i và j trong khoảng từ 0 đến M-1:
    - Nếu chưa bị xung đột thì thêm phần tử mới tại địa chỉ i này.

- Nếu bị xung đột thì hàm băm lại lần 1 f1 sẽ xét địa chỉ mới i+j, nếu lại bị xung đột thì hàm băm lại lần 2 f2 sẽ xét địa chỉ i+2j, …, quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử vào địa chi này.

- *Khi tìm kiếm* một phần tử có khoá key trong bảng băm, hàm băm i=f1(key) và j=f2(key) sẽ xác định địa chỉ i và j trong khoảng từ 0 đến M-1. Xét phần tử tại địa chỉ i, nếu chưa tìm thấy thì xét tiếp phần tử i+ji+2j, …, quá trình cứ thế cho đến khi nào tìm được khoá (trường hợp tìm thấy) hoặc bị rơi vào địa chỉ trống (trường hợp không tìm thấy).

Bảng băm dùng hai hàm băm khác nhau, hàm băm lại của phương pháp băm kép được tính theo I (từ hàm băm thứ nhất) và j (từ hàm băm thứ hai) theo một công thức bất kì, ở đây minh họa bằng địa chỉ mới cách j. Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.

Bảng băm với phương pháp băm kép nên chọn số địa chỉ M là số nguyên tố.

Bảng băm minh họa có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}
- Hàm băm f(key) = key % 10.

Minh hoạ:

Sau đây là minh hoạ cho bảng băm có tập khóa là tạâp số tự nhiên,tập địa chỉ có 11 địa chỉ (M=11)(từ địa chỉ 0 đến 10),chọn hàm băm f1(key)=key % 10 và f2(key)=9-key %9.

Xem việc minh hoạ này như một bài tập.

Cài đặt bảng băm dùng phương pháp băm kép:

a. Khai báo cấu trúc bảng băm:

```
#define NULLKEY –1
#define M 101
/*M la so nut co tren bang bam,du de chua cac nut
nhap vao bang bam,chon M la so nguyen to
*/
//Khai bao phan tu cua bang bam
struct node
{
        int key;//khoa cua nut tren bang bam
};

//khai bao bang bam co M nut
struct node hashtable[M];
int N;
//bien toan cuc chi so nut hien co tren bang bam
```

b. Các tác vụ:

Hàm băm:

Giả sử chúng ta chọn hai hàm băm dạng %:

f1(key0=key %M va f2(key) =M-2-key%(M-2).

```
//Ham bam thu nhat
int hashfunc(int key)
{
        return(key%M);
}
//Ham bam thu hai
int hashfunc2(int key)
{
        return(M-2 – key%(M-2));
}
```

Chúng ta có thể dùng hai hàm băm bất kỳ thay cho hai hàm băm dạng % trên.

Phép toán initialize :

Khởi động bảng băm.

Gán tất cả các phần tử trên bảng có trường key là NULL.

Gán biến toàn cục N = 0.

```
void initialize()
{
        int i;
        for (i = 0 ; i<M ; i++ )
                hashtable [i].key = NULLKEY;
        N = 0;// so nut hien co khoi dong bang 0
}
```

Phép toán empty :

Kiểm tra bảng băm có rỗng không.

```
int empty() .
{
        return (N == 0 ? TRUE : FALSE) ;
}
```

Phép toán full :

Kiểm tra bảng băm đã đầy chưa.

```
int full() .
{
        return (N == M-1 ? TRUE : FALSE) ;
}
```

Lưu ý bảng băm đầy khi N=M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.

Phép toán search :

Tìm kiếm phần tử có khóa k trên bảng băm, nếu không tìm thấy hàm này trả về về trị M, nếu tìm thấy hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
        int i, j ;
        i = hashfunc (k);
        j = hashfunc2 (k);
        While (hashtable [i].key!=k && hashtable [i] .key ! = NULLKEY)
        //bam lai (theo phuong phap bam kep)
                i = (i+j) % M ;
        if (hashtable [i]).key == k) // tim thay
                return (i) ;
        else// khong tim thay
        return (M) ;
}
```

Phép toán insert :

Thêm phần tử có khoá k vào bảng băm.

```
int insert(int k)
{
        int i, j;
        if (full () )
        {
                printf ("Bang bam bi day") ;
                return (M) ;
        }
        if (search (k) < M)
        {
                printf ("Da co khoa nay trong bang bam") ;
                return (M) ;
        }
        i = hashfunc (k) ;
        j = hashfunc 2 (k) ;
        while (hashtable [i].key ! = NULLEY)
                // Bam lai (theo phuong phap bam kep)
                i = (i + j) % M;
        hashtable [i].key = k ;
        N = N+1;
```

```
            return (i) ;
        }
```

Nhận xét bảng băm dùng phương pháp băm kép:

Nên chọn số địa chỉ M là số nguyên tố.

Bảng băm đầy khi N = M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.

Bảng băm được cài đặt theo cấu trúc này linh hoạt hơn bảng băm dùng phương pháp dò tuyến tính và bảng băm dùng phương pháp sò bậc hai, do dùng hai hàm băm khác nhau nên việc rải phần tử mang tính ngẫu nhiên hơn, nếu bảng băm chưa đầy tốc độ truy xuất có bậc O(1). Trường hợp xấu nhất là bảng băm gần đầy, tốc độ truy xuất chậm do thực hiện nhiều lần so sánh.

5. TỔNG KẾT VỀ PHÉP BĂM

- Bảng băm đặt cơ sở trên mảng.
- Phạm vi các giá trị khóa thường lớn hơn kích thước của mảng.
- Một giá trị khóa được băm thành một chỉ mục của mảng bằng hàm băm.
- Việc băm một khóa vào vào một ô đã có dữ liệu trong mảng gọi là sự đụng độ.
- Sự đụng độ có thể được giải quyết bằng hai phương pháp chính: Phương pháp nối kết và phương pháp băm lại.
- Trong phương pháp băm lại, các mục dữ liệu được băm vào các ô đã có dữ liệu sẽ được đưa vào ô khác trong mảng.
- Trong phương pháp nối kết, mỗi phần tử trong mảng có một danh sách liên kết. Các mục dữ liệu được băm vào các ô sẽ được đưa vào danh sách ở ô đó.

Vấn đề Hàm băm

- Hàm băm dùng phương pháp chia: h(k) = k mod m
- m là kích thước bảng băm, k là khóa.
- Hàm băm dùng phương pháp nhân: h(k) = $\lfloor m(k\ A \bmod 1) \rfloor$
- Knuth đề nghị A = 0.6180339887

Toàn bộ các phân tích của toàn bộ giải thuật
DPT KMP

AVL
BST
tại mỗi nút viết hàm đếm coi bao nhiêu phần tử là con của nó.

trên cây nhị phân tìm kiếm
cho trước 1 nút p, hay cho biết có bao nhiêu nút có giá trị nhỏ hơn giá trị của p

```
int timtrungvi( ref  root)
{
      n = demsophantu(root);
      x = timphantucosoluongphantunhohonlax(root, n/2);

}

int demSoPhanTuNhoHon(ref root, int x)
{
      if (root == null)
            return 0;
      if (root->key == x)
            return demSoPhanTuNhoHon(root->left, x);
      else
            if (root->key > x)
                  return demSoPhanTuNhoHon(root->left, x);
            else
            return 1 + demSoPhanTuNhoHon(root->left, x) +
demSoPhanTuNhoHon(root->right, x);
}
```

left, right
parent

Sort ==>
Tree ==>
Hash ==>
KMP ==>

Tính chất của từng thuật toán

Câu đầu tiên là 1 câu kinh điển

```
void insertnode(ref& p, ref q) q se dc chen vao cay root la p
{
      ref qleft = q->left, qright = q->right;
      q->left = null; q->right = null;
      insert(p,q);
      insertnode(p,qleft);
      insertnode(p, qright);
}

void createBST(ref &node)
{
ref newroot = null;

insertnode(newroot, root);
root = newroot;

}


struct NODE
{

NODE* prev;

NODE* next;

int   x;

NODE* myBestNextNode;

int     longestTail;


  x x x x x x x x x x x


}



for (NODE* p = tail; p != NULL; p = p->Prev)

{

      bestTail = 1;
```

```
for (NODE* q = p->Next; q!=NULL; q=q->Next)

      {
            if (q->x > p->x) // nối được
            {
                  if (bestTail < q->longestTail +1)
                  {
                        bestTail = q->longestTail + 1;
                        p->bestNextNode = q;
                  }
            }
      }
      p->longestTail = bestTail;
}

ket qua khong chac la head->bestTail

int res = 0;

for (p = head; p<=tail; p=p->Next)

      if (res<p->bestTail)

            res = p->bestTail;
```