BINARNA STABLA

Primer rekurzivnog obilaska stabla

- Odrediti zbir vrednosti svih listova zadatog binarnog stabla
- Izračunavanje obaviti korišćenjem rekurzivnog obilaska stabla

Primer rekurzivnog obilaska stabla

```
template <class T>
T BSTree<T>::sumLeafs(BSTNode<T> *pNode)
{
  if (pNode != NULL) {
      if (pNode->left == NULL
            && pNode->right == NULL) {
            return pNode->key;
      } else {
            return sumLeafs(pNode->left)
                  + sumLeafs(pNode->right);
      }
  } else {
      return 0;
  }
}
```

Primer rekurzivnog formiranja stabla

- Dat je niz od 2^n-1 elemenata za koje treba formirati potpuno binarno stablo
- Stablo popuniti elementima u postorder obilasku
- Za formiranje stabla koristiti rekurzivni obilazak

Primer rekurzivnog formiranja stabla

```
template <class T>
void BSTree<T>::createPostorder(BSTNode<T> *pNode, T **pElem,
   int level)
{
   if (level != 0) {
        pNode->left = new BSTNode<T>();
        pNode->right = new BSTNode<T>();
        createPostorder(pNode->left, pElem, --level);
        createPostorder(pNode->right, pElem, level);
        pNode->key = **pElem;
        (*pElem)++;
   } else {
        pNode->key = **pElem;
        (*pElem)++;
        pNode->left = NULL;
        pNode->right = NULL;
   }
}
```

Primer rekurzivnog formiranja stabla

```
template <class T>
void BSTree<T>::createTree(T niz[], int n)
{
  int level = 0;
  int tmp = n;
  while (tmp != 0) {
      tmp >>= 1;
      level++;
  }
  T **pElem = new T*();
  *pElem = niz;
  root = new BSTNode<T>();
  createPostorder(root, pElem, --level);
}
```

Određivanje sledbenika u uređenom binarnom stablu

- Odrediti sledbenika (po vrednosti) čvora u proizvoljnom uređenom binarnom stablu

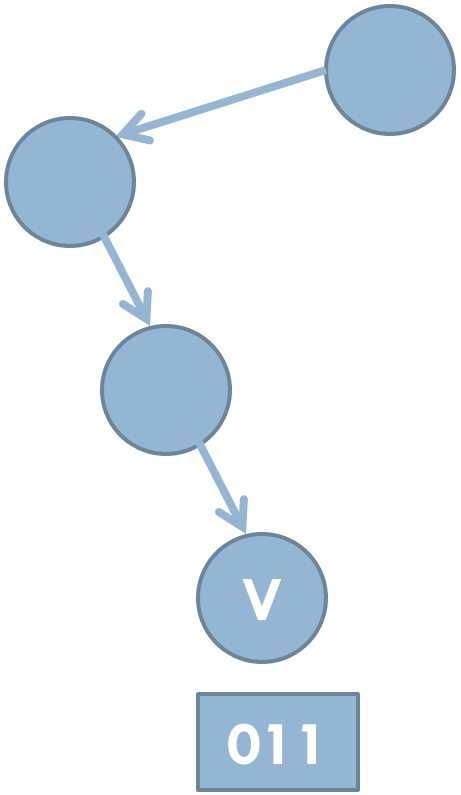
Određivanje sledbenika u uređenom binarnom stablu

```
template <class T>
T BSTree<T>::successorNode(T elem) {
   setParents(root, NULL);
   BSTNode<T> *pNode = search(elem);
   if (pNode->right != NULL) {
        pNode = pNode->right;
        while (pNode->left != NULL)
                pNode = pNode->left;
   } else {
        BSTNode<T> *pPar = pNode->par;
        while (pPar != NULL && pPar->right == pNode) {
                pNode = pPar;
                pPar = pNode->par;
        }
        pNode = pPar;
   }
   if (pNode != NULL)
        return pNode->key;
   else
        return T();
}
```

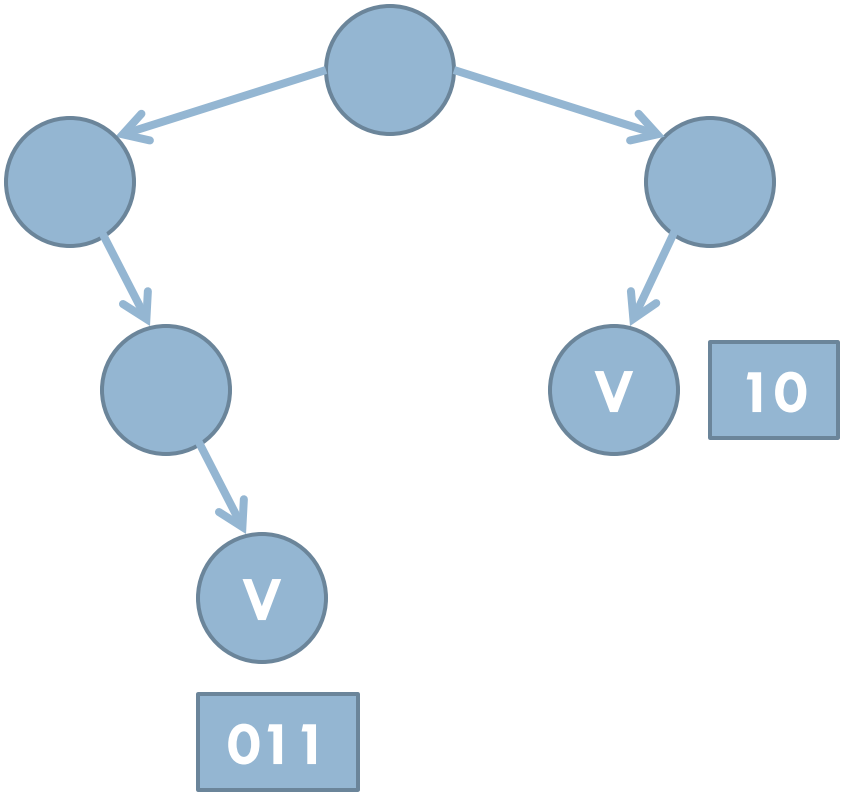
Radix stablo

- Stringa $a=a_0a_1\ldots a_p$ je leksikografski manji od stringa $b=b_0b_1\ldots b_q$ ako važi:
  - Postoji ceo broj j za koji važi da je $0\leq j\leq \min(p,q)$, pri čemu je $a_i=b_i$ za svako i=0,1,…,j-1 i , ili
  - $p<q$ i $a_i=b_i$, za svako i=0,1,…,p.
- Na primer: 10100 < 10110 10100 < 101000
- Radix stablo se formira tako što za svaki string koji se dodaje u stablo odgovarajući čvor stabla ima vrednost validan, u suprotnom ne.
- Grane stabla predstavljaju elemente stringa, pri čemu leva grana ima vrednost 0, a desna 1.

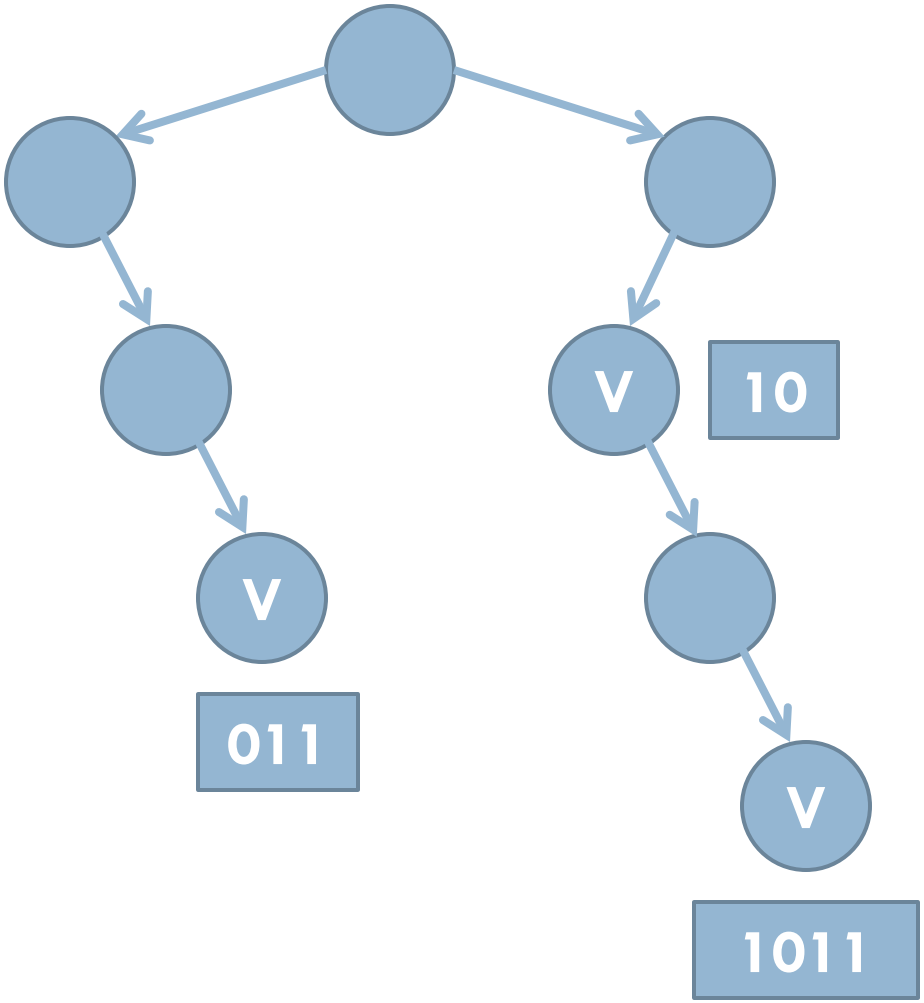
Radix stablo



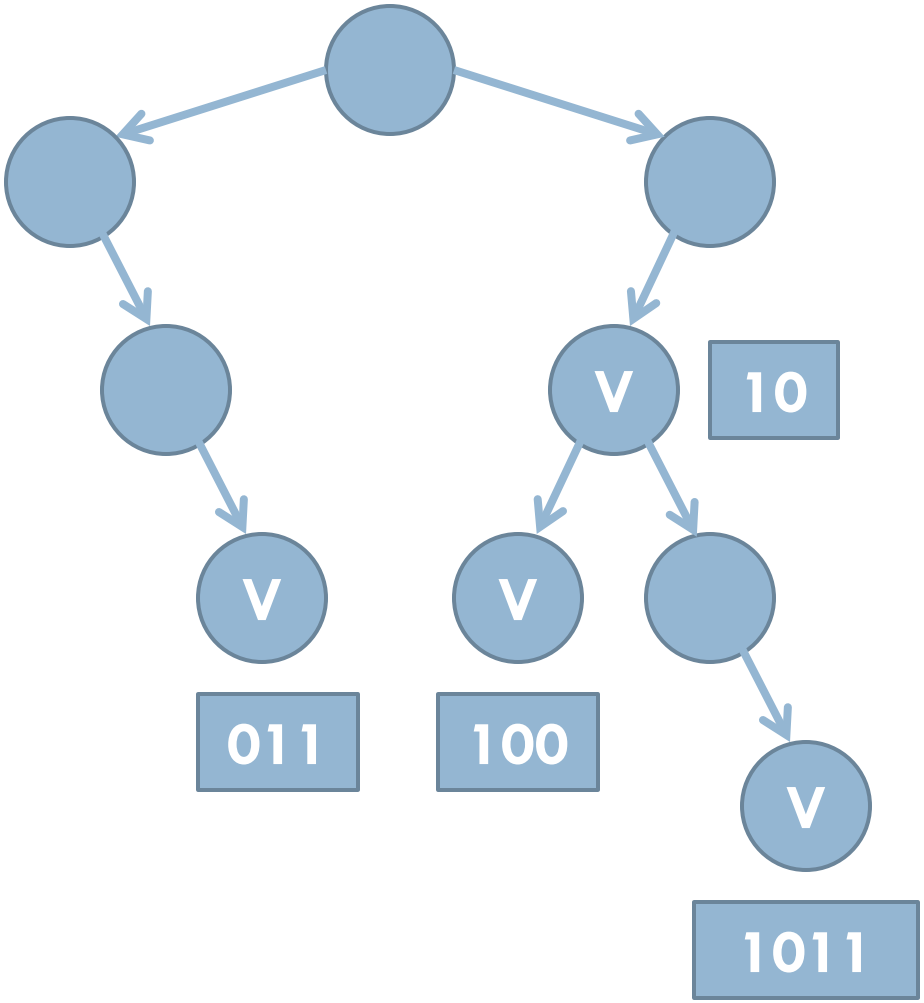
Radix stablo



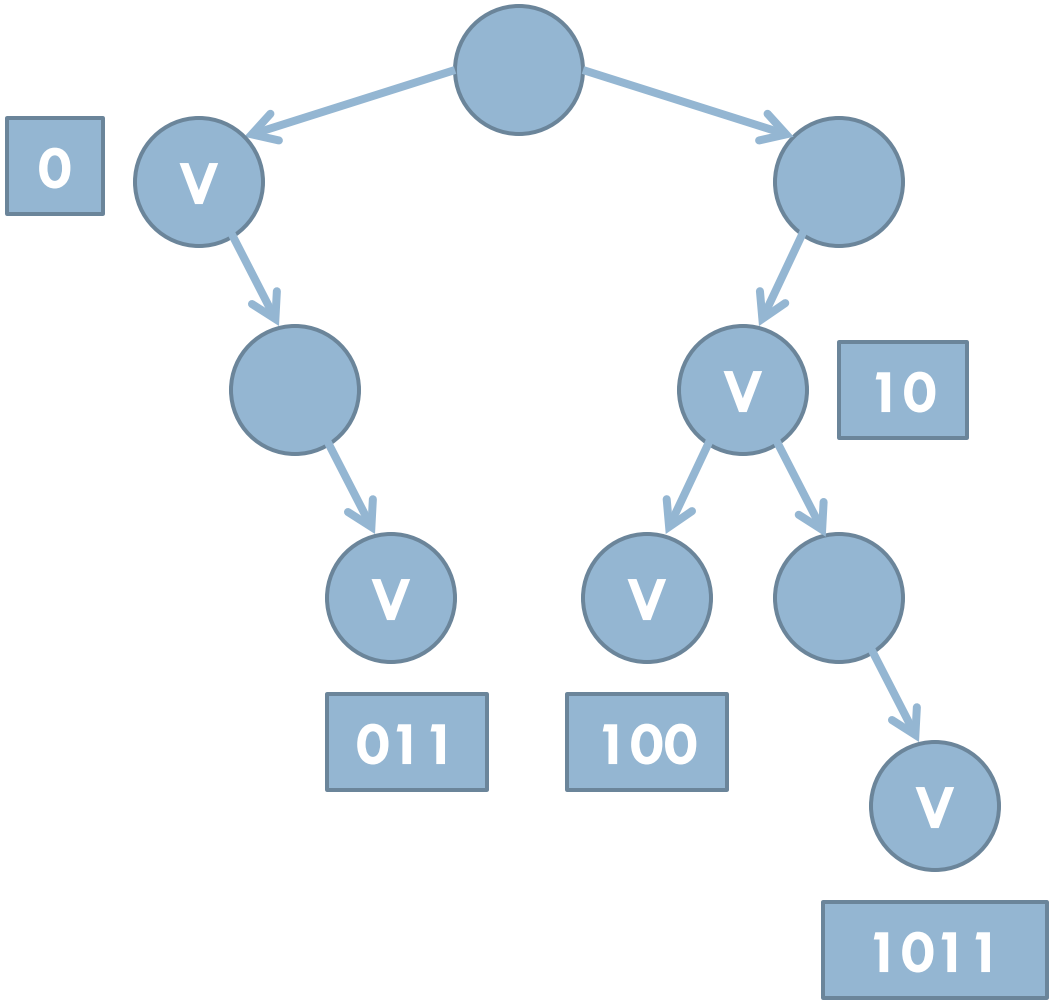
Radix stablo



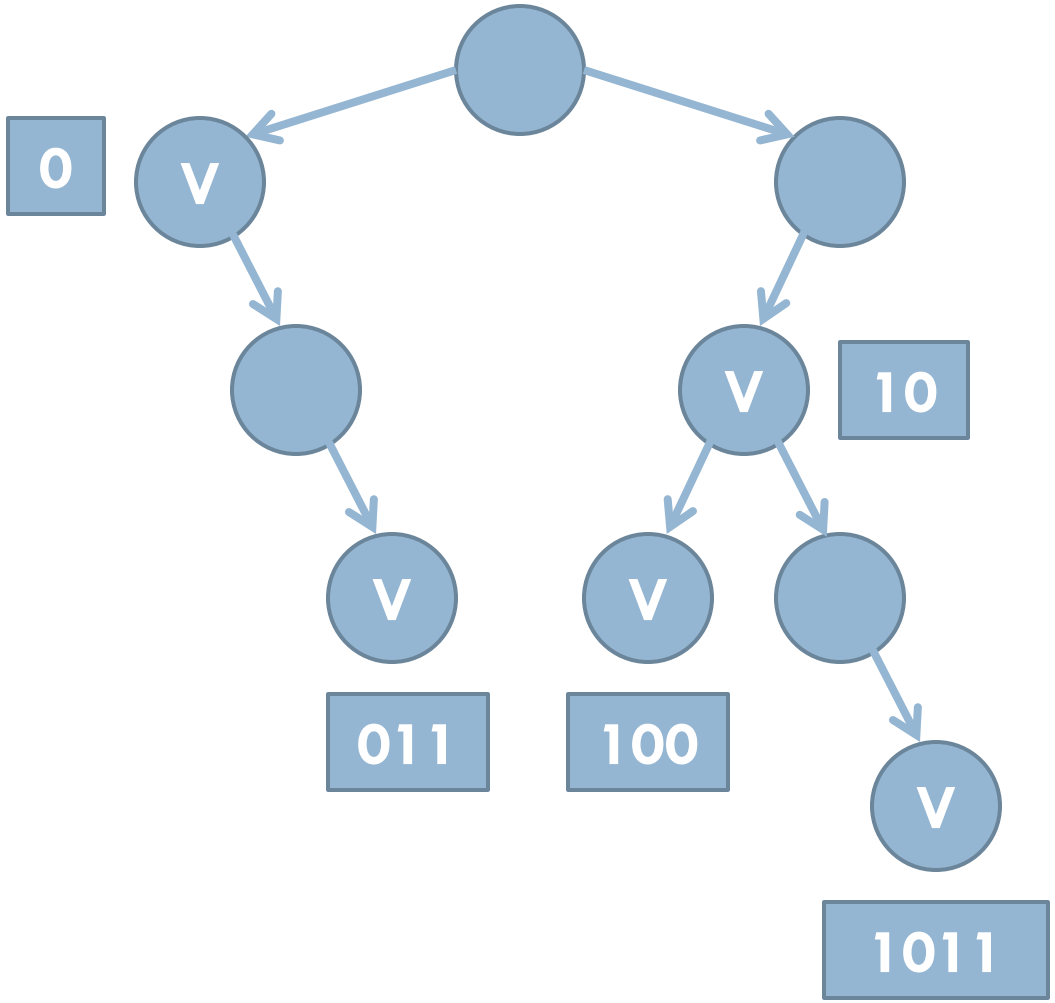
Radix stablo



Radix stablo



Radix stablo



- 0
- 011
- 10
- 100
- 1011

Radix stablo

```
template <class T>
void BSTree<T>::createRadix(BSTNode<T> *pNode, char data[], int ind)
{
    if (ind < strlen(data)) {
          BSTNode<T> *pChild;
          if (data[ind] == '0') {
                    pChild = pNode->left;
                    if (pChild == NULL) {
                              pChild = new BSTNode<T>();
                              pChild->key = 0;
                              pNode->left = pChild;
                    }
          } else {
                    pChild = pNode->right;
                    if (pChild == NULL) {
                              pChild = new BSTNode<T>();
                              pChild->key = 0;
                              pNode->right = pChild;
                    }
          }
          createRadix(pChild, data, ++ind);
    } else {
          pNode->key = 1;
    }
}
```

Radix stablo

```
template <class T>
void BSTree<T>::createRadixTree(char* niz[], int n)
{
   root = new BSTNode<T>();
   root->key = 0;
   int i;
   for (i=0; i<n; i++) {
        createRadix(root, niz[i], 0);
   }
}

template <class T>
void BSTree<T>::printRadix()
{
   char data[10];
   printRadix(root, data, 0);
}
```

Radix stablo

```
template <class T>
void BSTree<T>::printRadix(BSTNode<T> *pNode,
                                  char data[], int ind)
{
  if (pNode != NULL) {
      if (pNode->key == 1) {
            data[ind] = '\0';
            cout << data << " ";
      }
      data[ind] = '0';
      printRadix(pNode->left, data, ind+1);
      data[ind] = '1';
      printRadix(pNode->right, data, ind+1);
  }
}
```

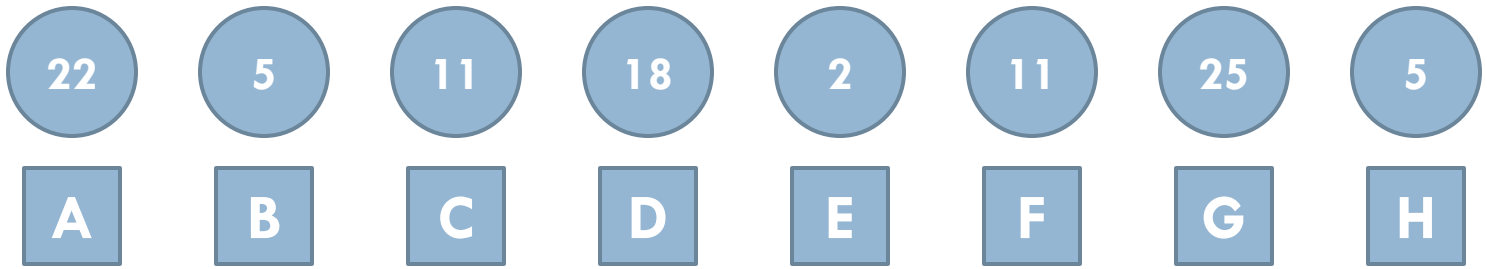
Huffman-ovi kodovi

- Koriste se za kodiranje karaktera kako bi se smanjila količina podataka koja se dobija kodiranjem
- Obezbeđuju uštedu od 20%-90%
- Huffman-ov kod se formira na osnovu učestalosti pojavljivanja karaktera koje treba kodirati
- Ideja: Karaktere koji se više puta pojavljuju kodirati kraćim rečima (nizom bitova)
- Huffman-ov kod se formira na osnovu binarnog stabla u čijim listovima se nalaze karakteri

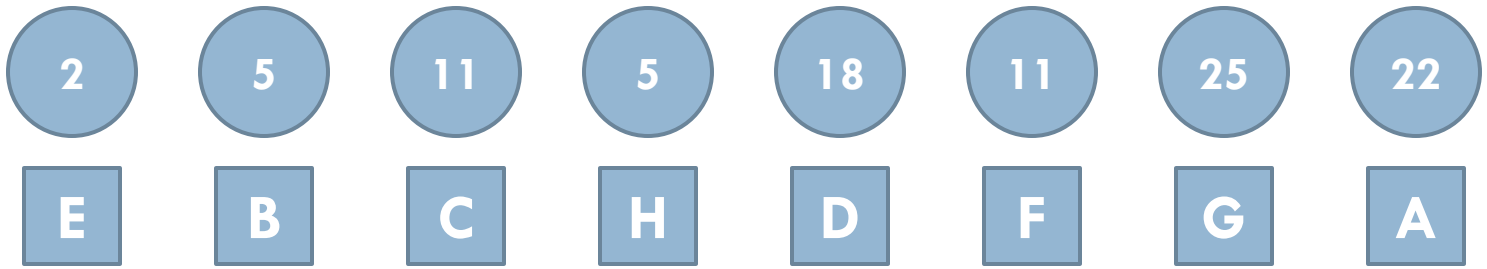
Huffman-ovi kodovi

- Stablo se formira rekurzivno od listova ka korenu
- Na početku svi karakteri su stabla za sebe
- U svakoj iteraciju se biraju dva stabla čiji koreni imaju najmanju učestanost pojavljivanja i ova dva stabla se spajaju u novo, tako da koren sadrži zbir učestanosti korena ova dva stabla koja postaju njegovi potomci
- Postupak spajanja se završava kada ostane samo jedno stablo koje predstavlja rezultat
- Kod karaktera se određuje na osnovu puta u stablu od korena do lista gde je smešten karakter, pri čemu levi link predstavlja 0, a desni 1.

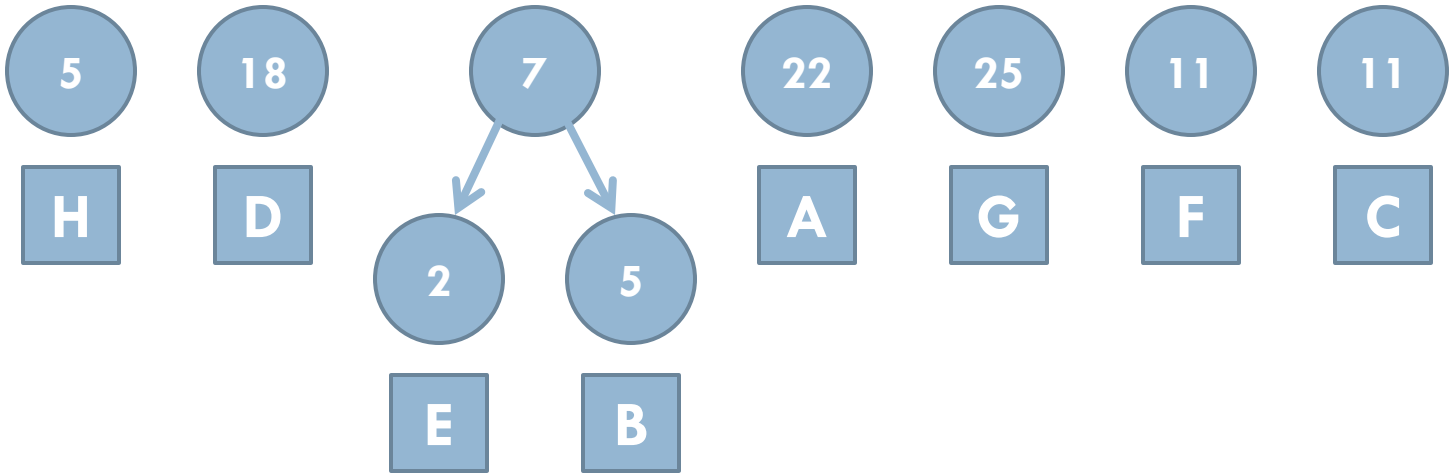
Huffman-ovi kodovi



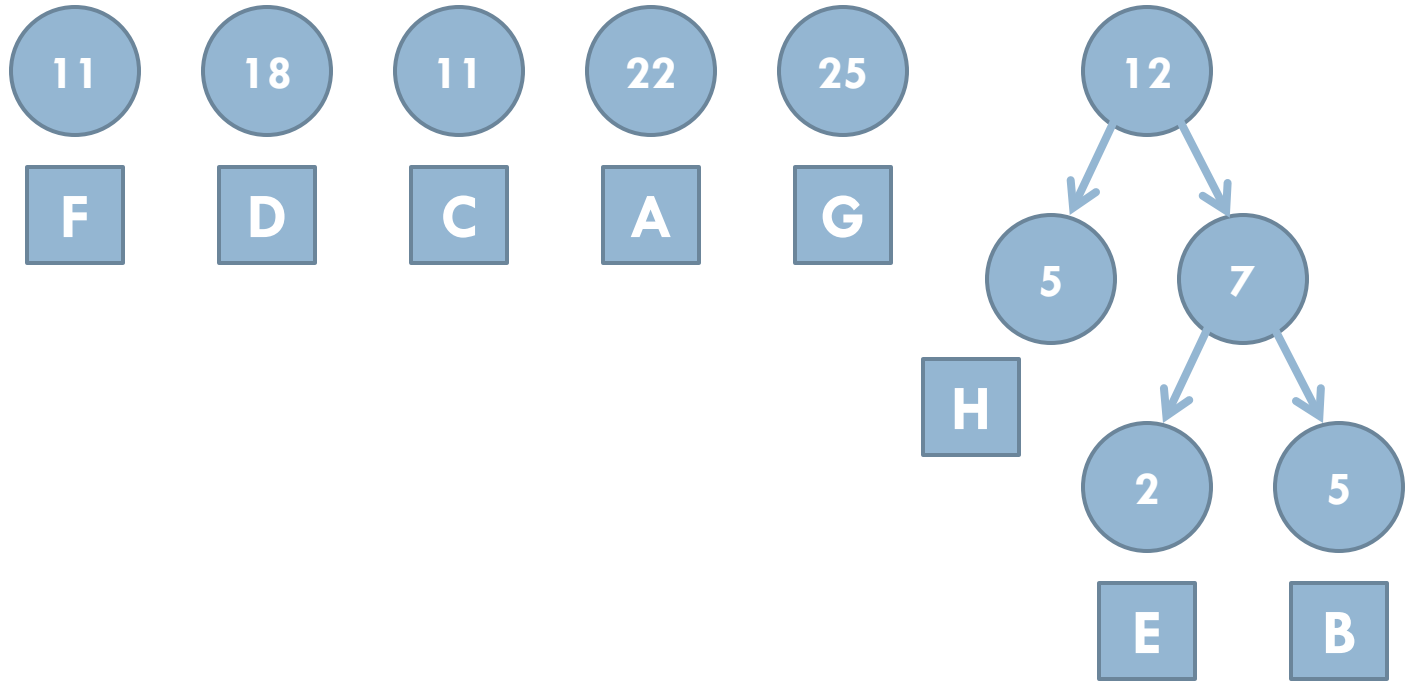
Huffman-ovi kodovi



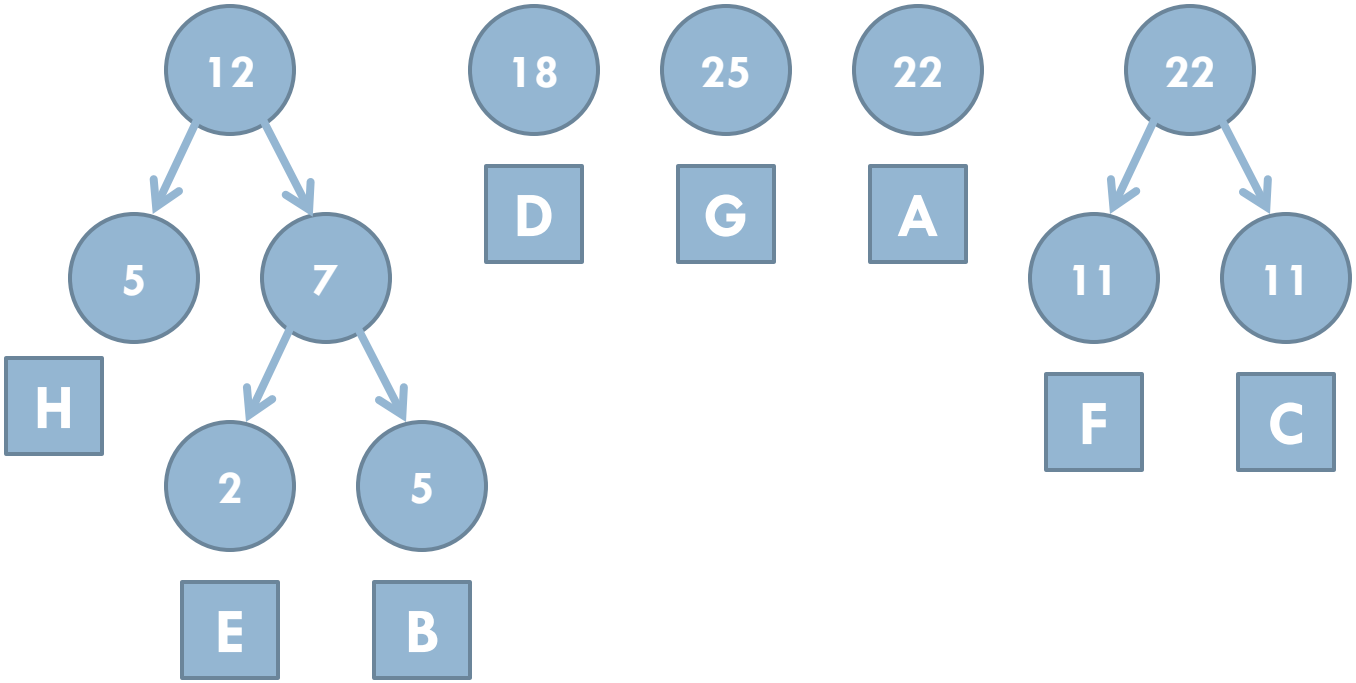
Huffman-ovi kodovi



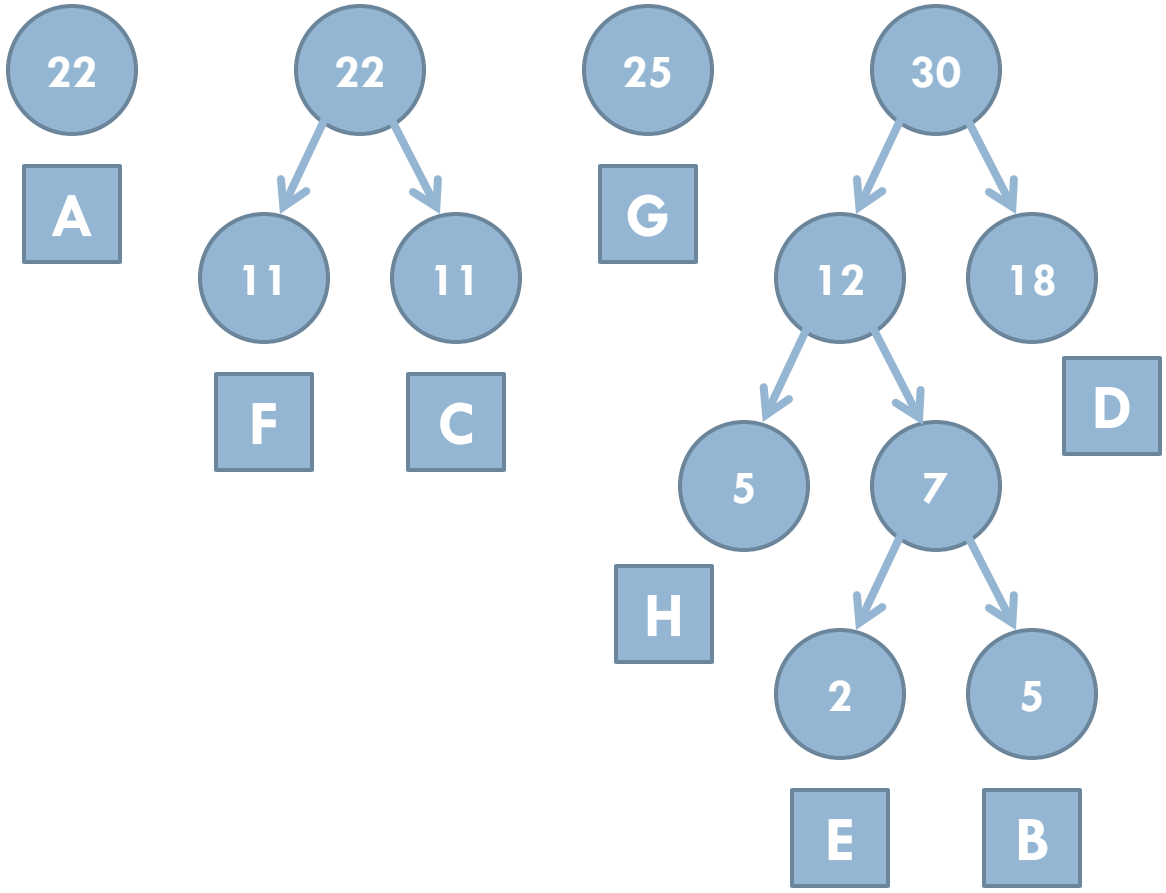
Huffman-ovi kodovi



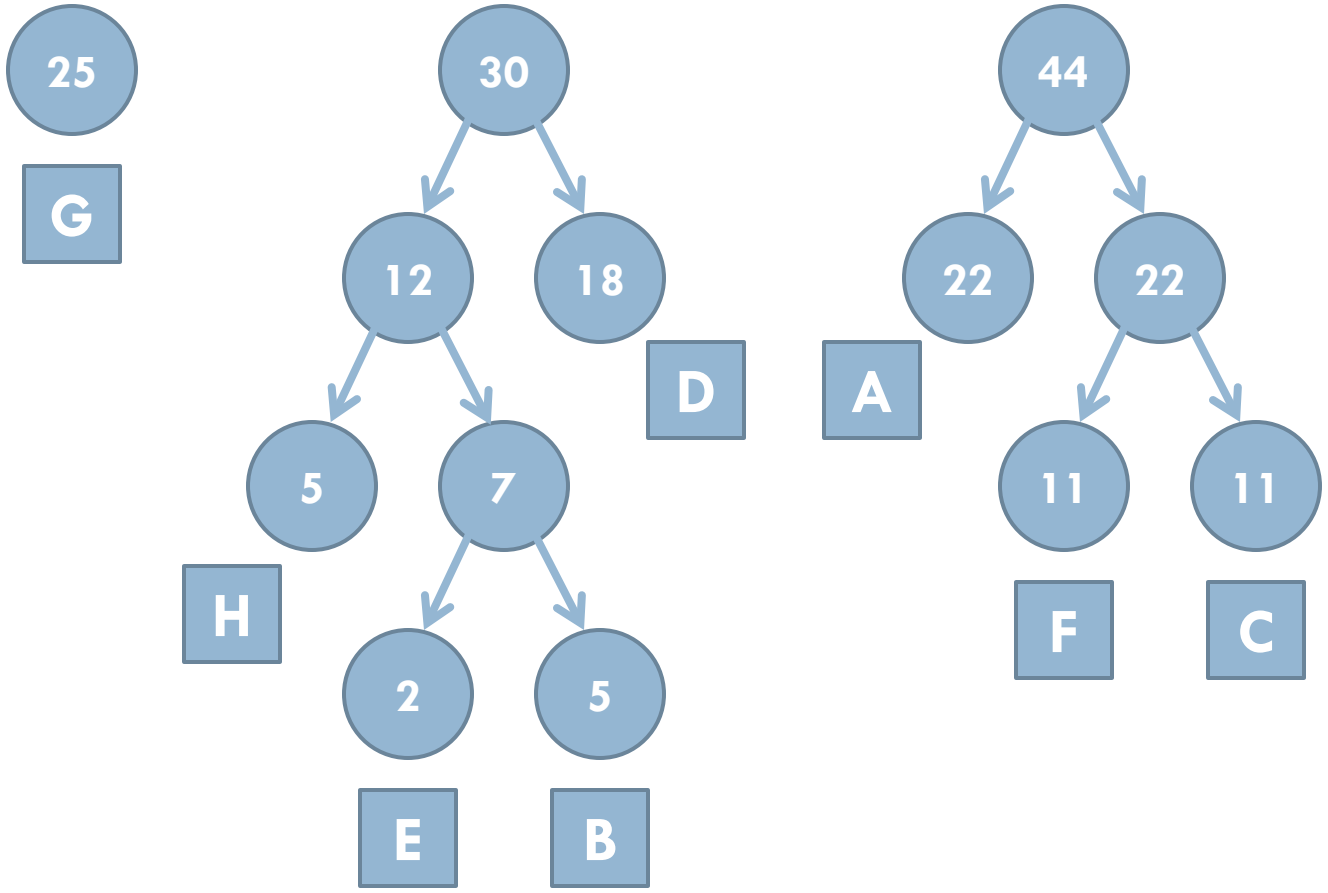
Huffman-ovi kodovi



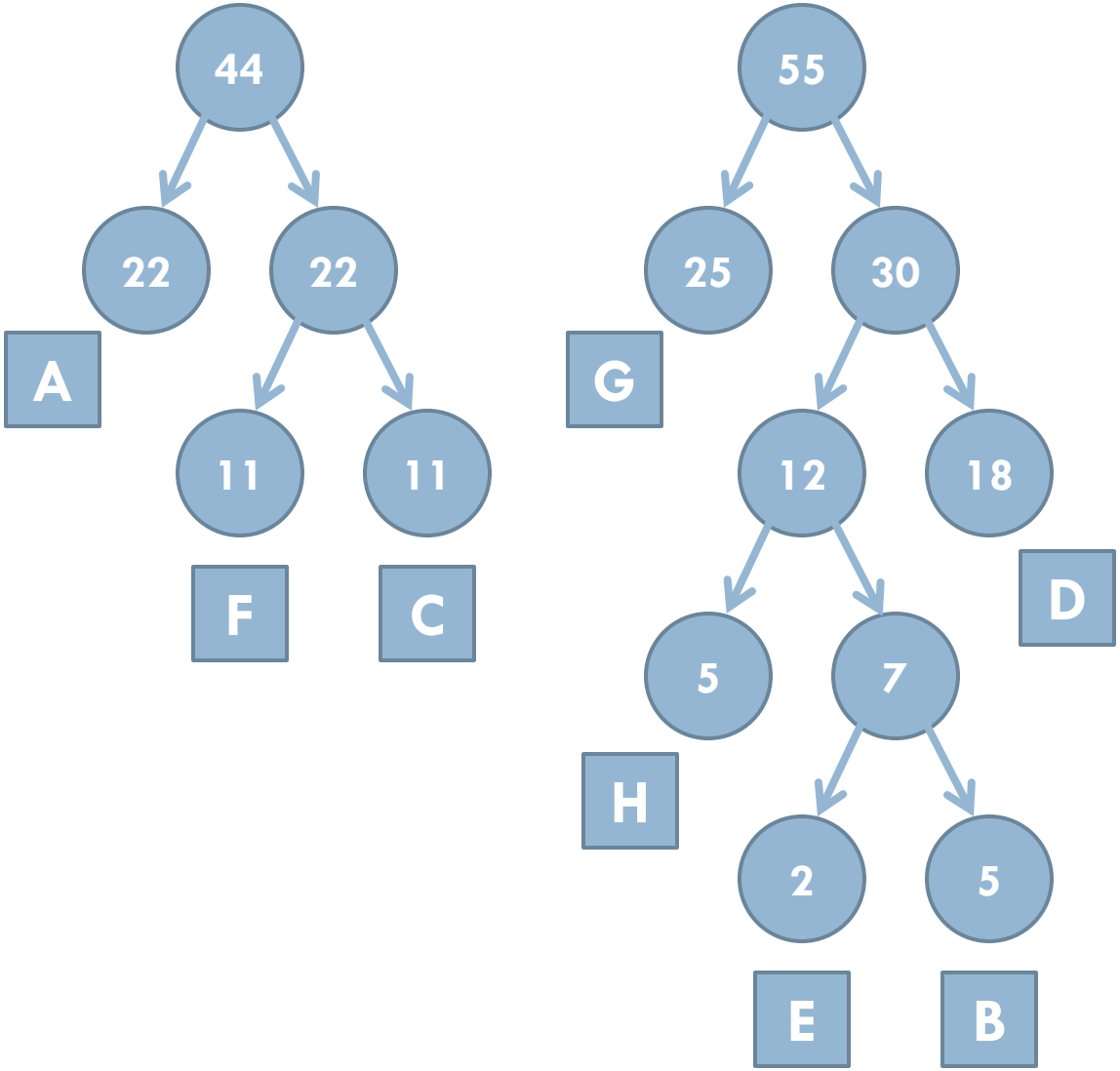
Huffman-ovi kodovi



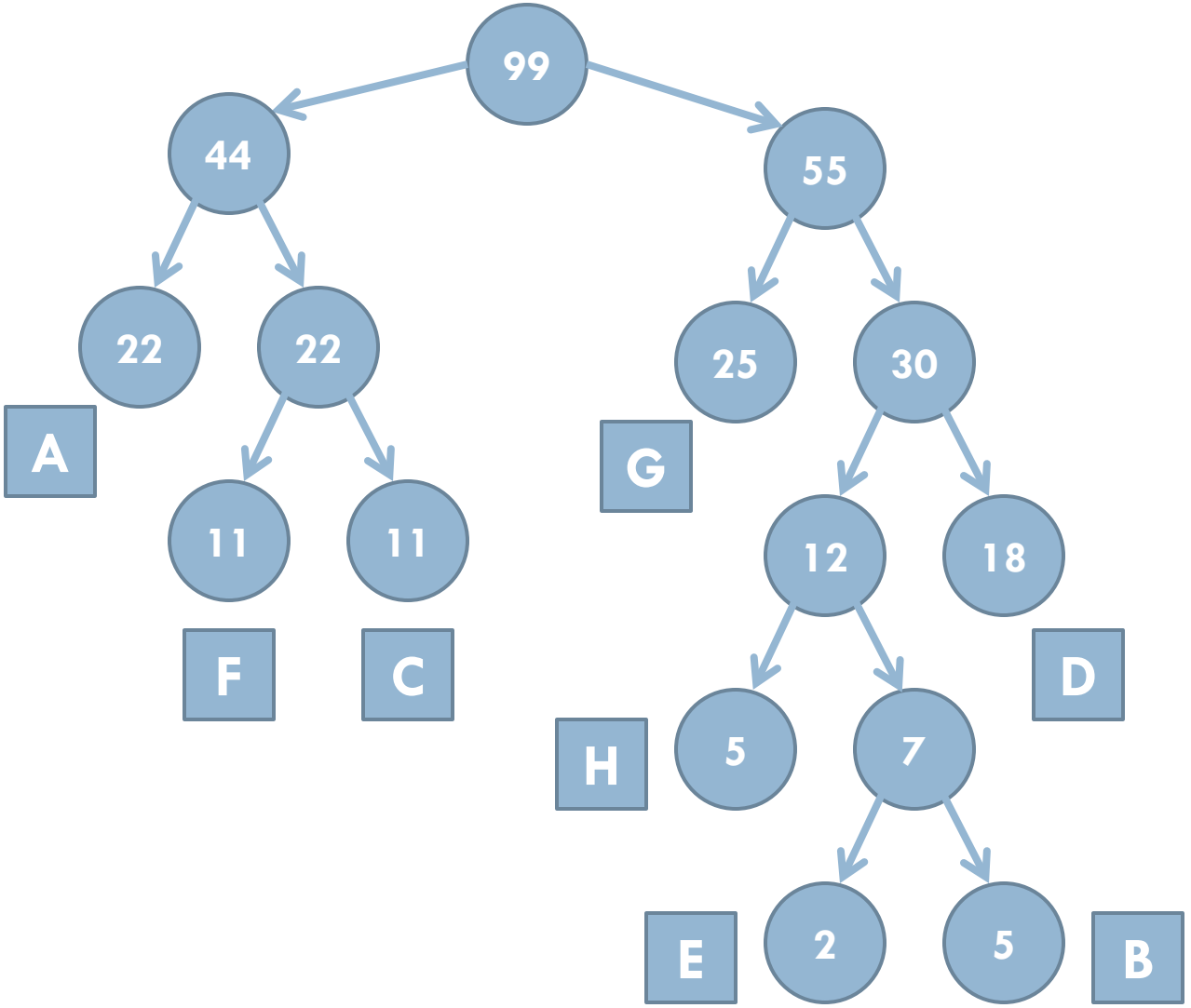
Huffman-ovi kodovi



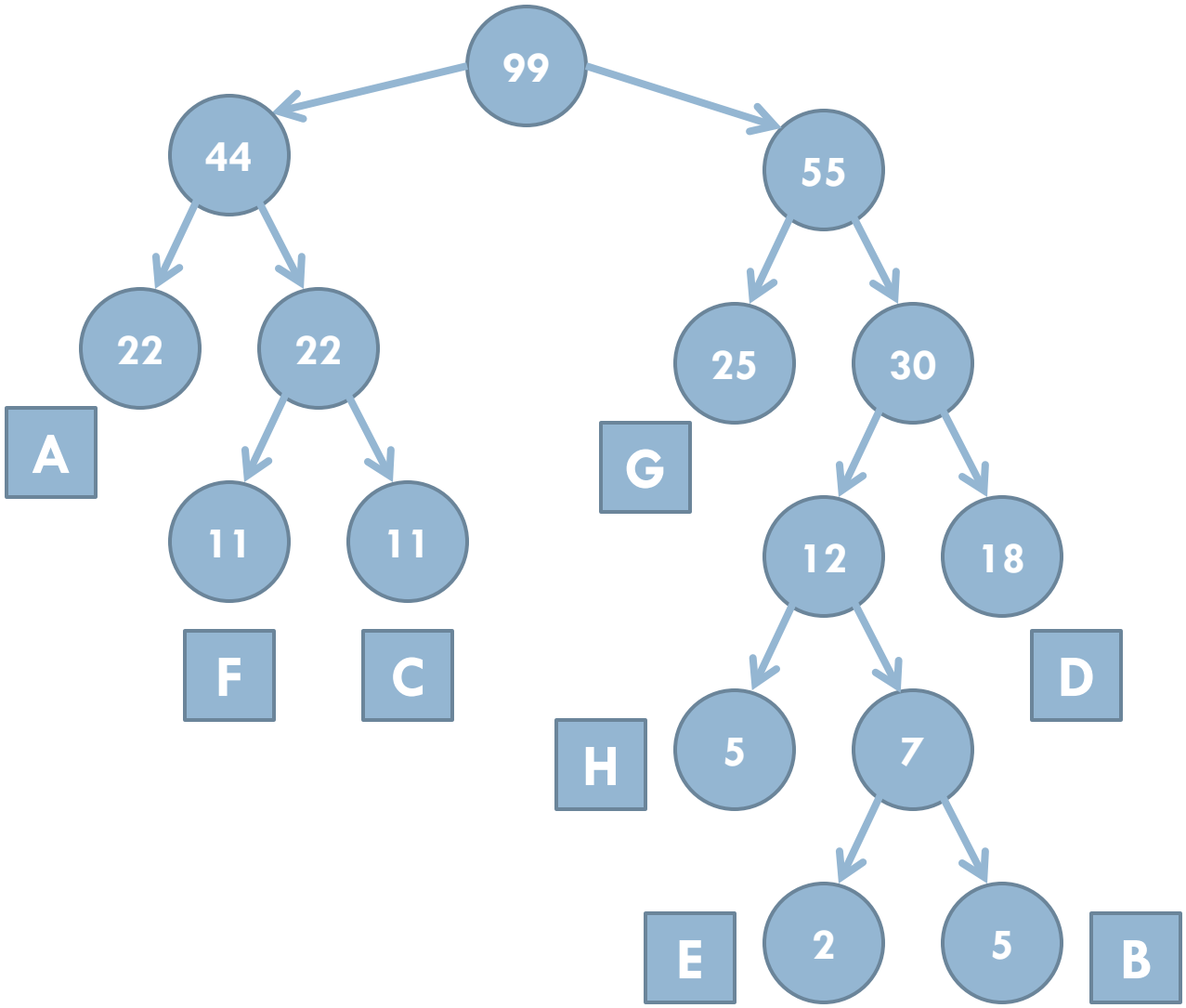
Huffman-ovi kodovi



Huffman-ovi kodovi



Huffman-ovi kodovi



A – 00
B – 11011
C – 011
D – 111
E – 11010
F – 010
G – 10
H – 1100

Huffman-ovi kodovi

```
class HuffElem {
public:
   char sign;
   int freq;
   HuffElem() { sign = '\0'; freq = 0; };
   HuffElem(char s, int f) { sign = s; freq = f; };
   HuffElem operator + (const HuffElem& sec) {
        HuffElem huff; huff.sign='\0'; huff.freq=freq+sec.freq;
        return huff; };
   bool operator < (const HuffElem& huff) {
        return freq < huff.freq; };
   bool operator <= (const HuffElem& huff) {
        return freq <= huff.freq; };
   bool operator > (const HuffElem& huff) {
        return freq > huff.freq; };
};
```

Huffman-ovi kodovi

```
template <class T>
class HuffFreq {
   BSTNode<T> *pNode;
public:
   HuffFreq() { pNode = NULL; };
   HuffFreq(BSTNode<T> *pN) { pNode = pN; };
   HuffFreq(HuffFreq<T>& huff) { pNode = huff.pNode; };
   HuffFreq<T>& operator = (const HuffFreq<T>& huff) {
        pNode = huff.pNode; return *this; };
   bool operator < (const HuffFreq<T>& huff) {
        return pNode->key < huff.pNode->key; };
   bool operator <= (const HuffFreq<T>& huff) {
        return pNode->key <= huff.pNode->key; };
   bool operator > (const HuffFreq<T>& huff) {
        return pNode->key > huff.pNode->key; };
   BSTNode<T>* getValue() { return pNode; };
};
```

Huffman-ovi kodovi

```
template <class T>
void BSTree<T>::createHuffman(HuffElem aHuff[], int n)
{
   BinaryMinHeap< HuffFreq<T> > heapHuff(n+1);
   int i;
   for (i=0; i<n; i++) {
        BSTNode<T> *pNode = new BSTNode<T>();
        pNode->key = aHuff[i];
        HuffFreq<T> hufFreq(pNode);
        heapHuff.insert(hufFreq);
   }
   for (i=0; i<n-1; i++) {
        BSTNode<T> *pNode = new BSTNode<T>();
        pNode->left = heapHuff.deleteRoot().getValue();
        pNode->right = heapHuff.deleteRoot().getValue();
        pNode->key = pNode->left->key + pNode->right->key;
        HuffFreq<T> hufFreq(pNode);
        heapHuff.insert(hufFreq);
   }
   root = heapHuff.deleteRoot().getValue();
}
```