Algorithms and Data Structures 2

16 – B-trees

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **Definition**
- **Motivation**
- **Properties**
- **Operations**
  - Search
  - Insertions
- **Variants**
- **Applications**

B-trees

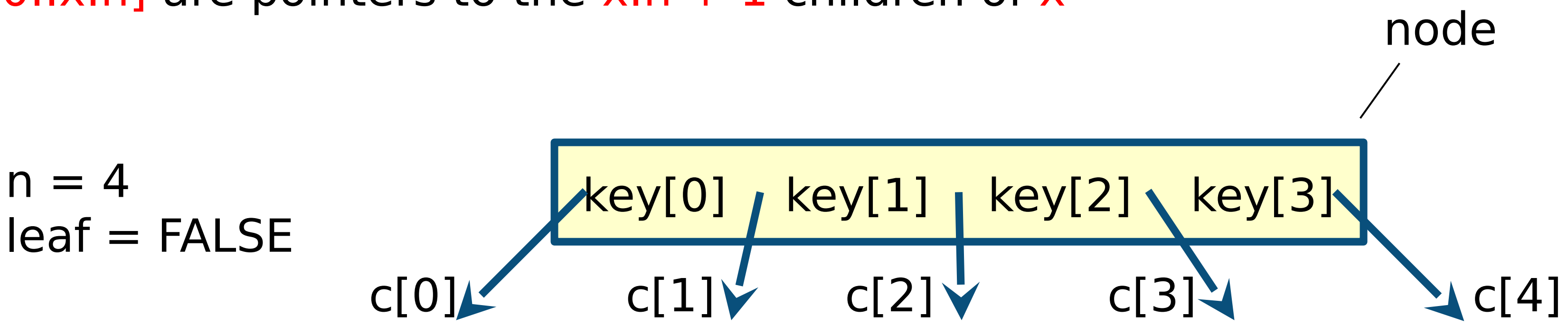
- **Balanced search trees introduced by Bayer in 1972**
  - Same inventor of red-black trees!
  - Bayer, Rudolf, and Edward M. McCreight. "Organization and Maintenance of Large Ordered Indices." Acta Informatica 1 (1972): 173-189.

- **Designed for big data sets stored on disks or other secondary storage devices**
  - Number of I/O operations is minimised

- **B-tree nodes may have more than two children**
  - Typically thousands depending on the physical characteristic of the hard disk
  - Height is O(log n) but usually much less than that of a red-black tree (example later in this lecture)

Definition

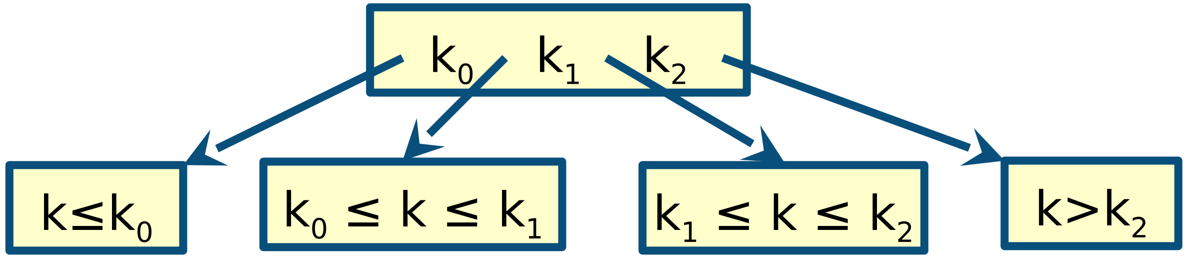
- **A node x in a B-tree has the following attributes**
  - x.n is the number of keys stored in the node
  - x.key[0..x.n-1] are the x.n keys stored in nondecreasing order
  - x.leaf is TRUE if x is a leaf, FALSE otherwise
  - x.c[0..x.n] are pointers to the x.n + 1 children of x



- **An attribute T.root points to the root of B-tree T**
- **We call a fixed integer $t \geq 2$ the minimum degree (number of children) of a B-tree**

Definition (cont.)

- **A B-tree is a rooted tree satisfying the following properties**
  1. Leaves have no children (x.c is NIL)
  2. The keys separate the ranges of keys stored in each subtree (see example below)



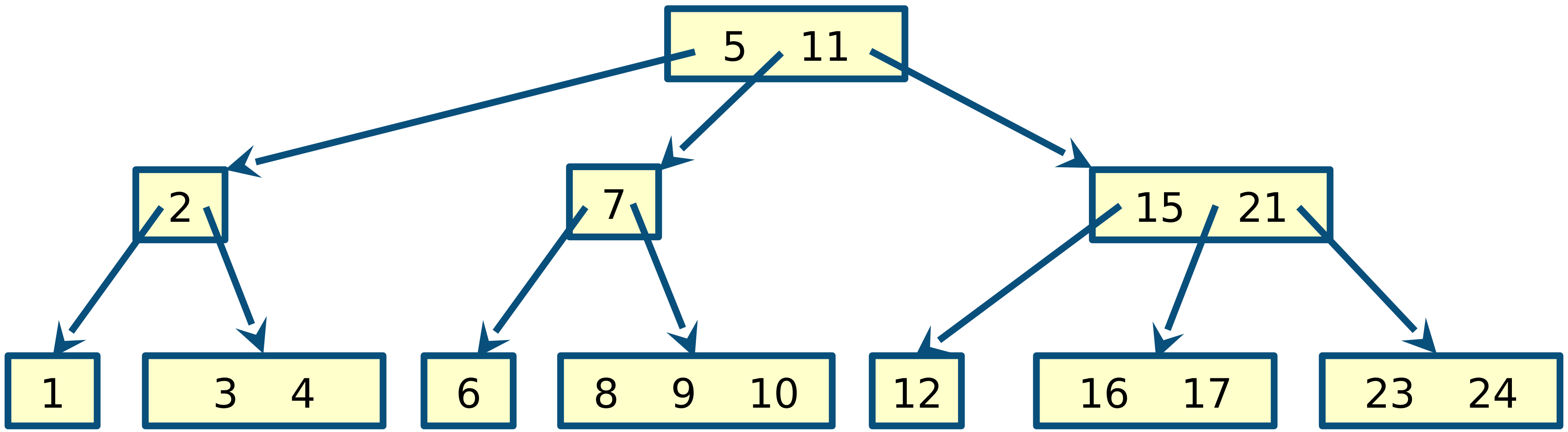
  3. All leaves have the same depth (which is the tree height)
  4. Every node other than the root must have at least t - 1 keys
  5. Every node may contain at most 2t - 1 keys

Bounds on the number of keys

- **A node is full when it contains 2t - 1 keys**

Example

- **Minimum degree is t = 2**
  - Sometimes called 2-3-4 trees, as every internal node has either 2, 3, or 4 children
- **The tree stores 18 keys and has height 2**
  - A red-black tree of height 5 is required to store 18 keys



full node

Worst-case height of a B-tree

- **If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$**

- **Proof**
  - The root of T contains at last one key, while all other nodes have at least $t - 1$ keys
  - Therefore, T has at least 2 nodes at depth 1, $2t$ nodes at depth 2, $2t^2$ nodes at depth 3, …
  - In general, we have $2t^{h-1}$ nodes at depth h
  - The total number of nodes is obtained by summing the nodes at each depth
  - Therefore, the number of keys n satisfies the following inequality

Geometric series

  - Take the base-t log on both sides to conclude the proof

Motivation for B-trees

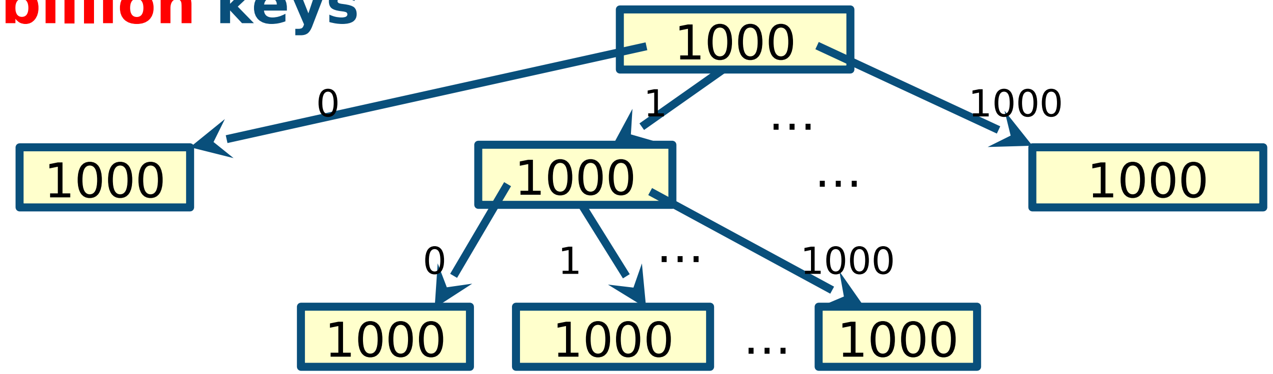
- **When a data structure is stored in secondary memory accessing a single node may take up to 8-10 milliseconds**
  - More than 100k memory accesses can be performed in the same time span
  - Primary memory access time is 50 nanoseconds

- **Each access retrieves a page of bits (2^{11} – 2^{14} bytes) not just one item to amortise the time spent waiting for mechanical movements**

- **To minimise the number of disk access, B-trees minimise the number of nodes**
  - Height is reduced by allowing high branching factors
  - Bigger nodes, slower to process, but faster overall as disk access is very slow

Example

- **A B-tree with branching factor 1001 and height 2 can store over one billion keys**



1 node	1000 keys
1001 nodes	$1001 * 10^3$ keys
1001^2 nodes	$1001^2 * 10^3$ keys

- **Keeping the root node in memory allows us to access any key by making at most 2 disk accesses**

- **To minimise I/O, a B-tree node is usually as large as a whole disk page**
  - Typical branching factors range from 50 to 2000

Modelling I/O

- **Typically the amount of data to be stored in a B-tree is so large that all the data cannot fit into main memory at once**
  - The B-tree algorithms copy selected pages from disk to main memory as needed and write back onto disk the pages that have changed
  - We assume the operating system flushes from main memory pages no longer in use

- **We explicitly model disk operations in the pseudocode**
  - DISK-READ(x) reads object x into main memory
  - DISK-WRITE(x) save object x onto disk
  - ALLOCATE-NODE() allocates one disk page in O(1) time

Operations

- **The root of the B-tree is always stored in the main memory**
  - No DISK-READ on the root needed
  - A DISK-WRITE of the root is needed whenever the root is modified
- **Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them**

- **We study the one-pass version of the B-tree operations**
  - Proceed downward from the root of the tree, without having to back up
  - SEARCH
  - CREATE
  - INSERT
  - DELETE is not part of the course

Search

- **Similar to recursive BST search**
  - Instead of doing a binary decision at each node (left/right), we do a multiway decision according to the number of children of the node

```
SEARCH(x,k)
  i := 0
  while i < x.n and k > x.key[i]
    i := i + 1
  if i < x.n and k = x.key[i]
    return (x,i)
  elseif x.leaf
    return NIL
  else
    DISK-READ(x,c[i])
    return SEARCH(x.c[i],k)
```

- **Top level call is SEARCH(T.root,k)**

- **Return a node x and an index i such that x.key[i] = k**
- **Return NIL if k not found**

Analysis

- **n is the number of keys and t is the minimum degree**

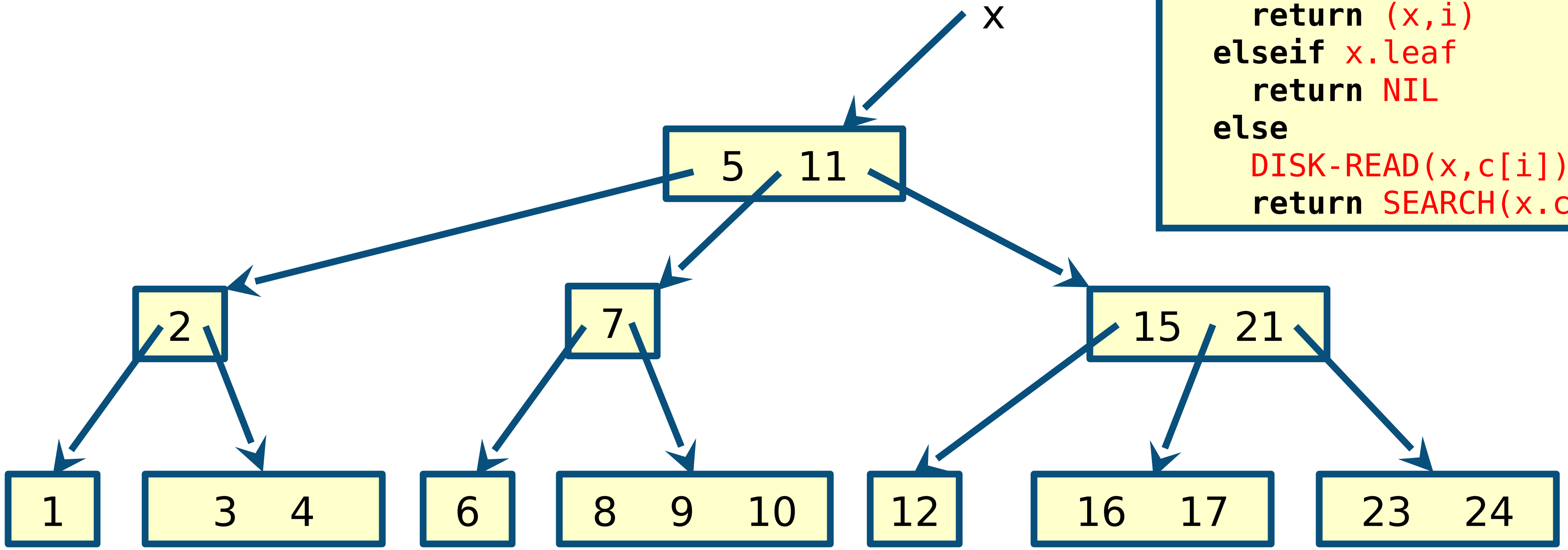
- **SEARCH performs $O(\log_t n)$ disk accesses**

- **Total CPU time is $O(t \log_t n)$**
  - While loop takes $O(t)$ time since $x.n < 2t$

```
SEARCH(x,k)
  i := 0
  while i < x.n and k > x.key[i]
    i := i + 1
  if i < x.n and k = x.key[i]
    return (x,i)
  elseif x.leaf
    return NIL
  else
    DISK-READ(x,c[i])
    return SEARCH(x.c[i],k)
```

Example

- **Find key 16**
  - x = T.root

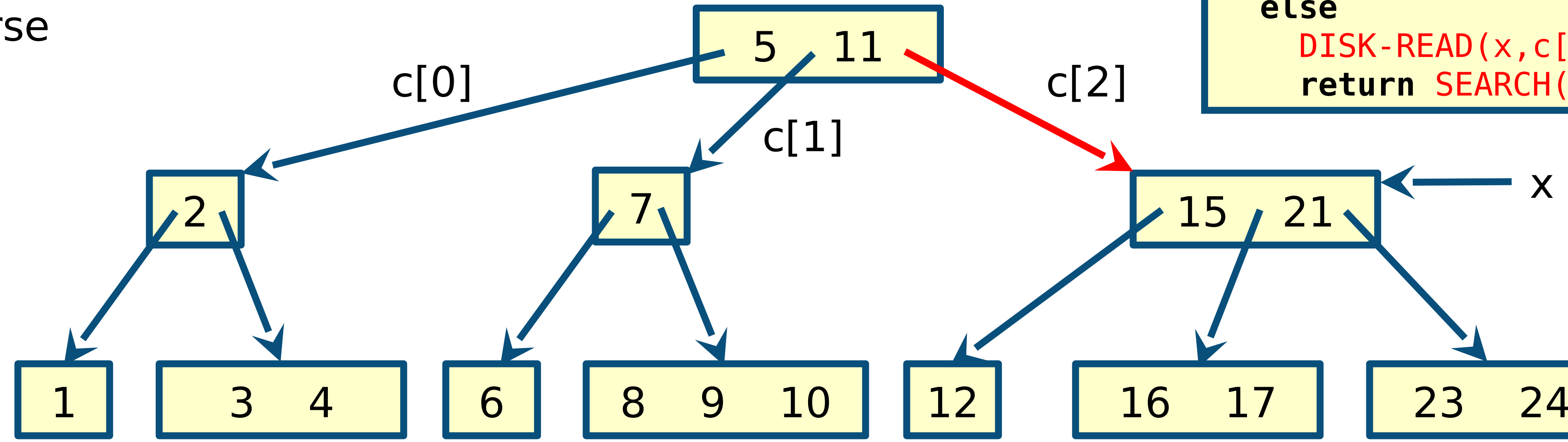



```
SEARCH(x,k)
  i := 0
  while i < x.n and k > x.key[i]
    i := i + 1
  if i < x.n and k = x.key[i]
    return (x,i)
  elseif x.leaf
    return NIL
  else
    DISK-READ(x,c[i])
    return SEARCH(x.c[i],k)
```

Example

- **Find key 16**
  - 16 > 5 and 16 > 11
  - i = 2
  - Retrieve next node from disk
  - Recourse

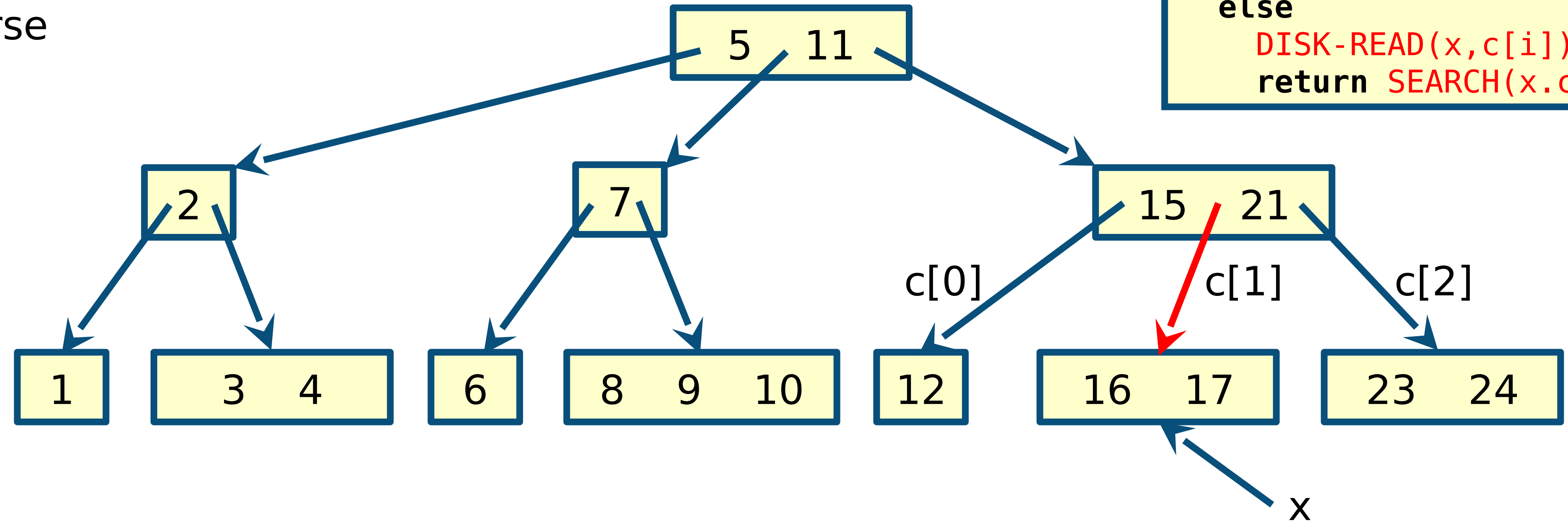
```
SEARCH(x,k)
  i := 0
  while i < x.n and k > x.key[i]
    i := i + 1
  if i < x.n and k = x.key[i]
    return (x,i)
  elseif x.leaf
    return NIL
  else
    DISK-READ(x,c[i])
    return SEARCH(x.c[i],k)
```



Example

- **Find key 16**
  - 15 < 16 < 21
  - i = 1
  - Retrieve next node from disk
  - Recourse

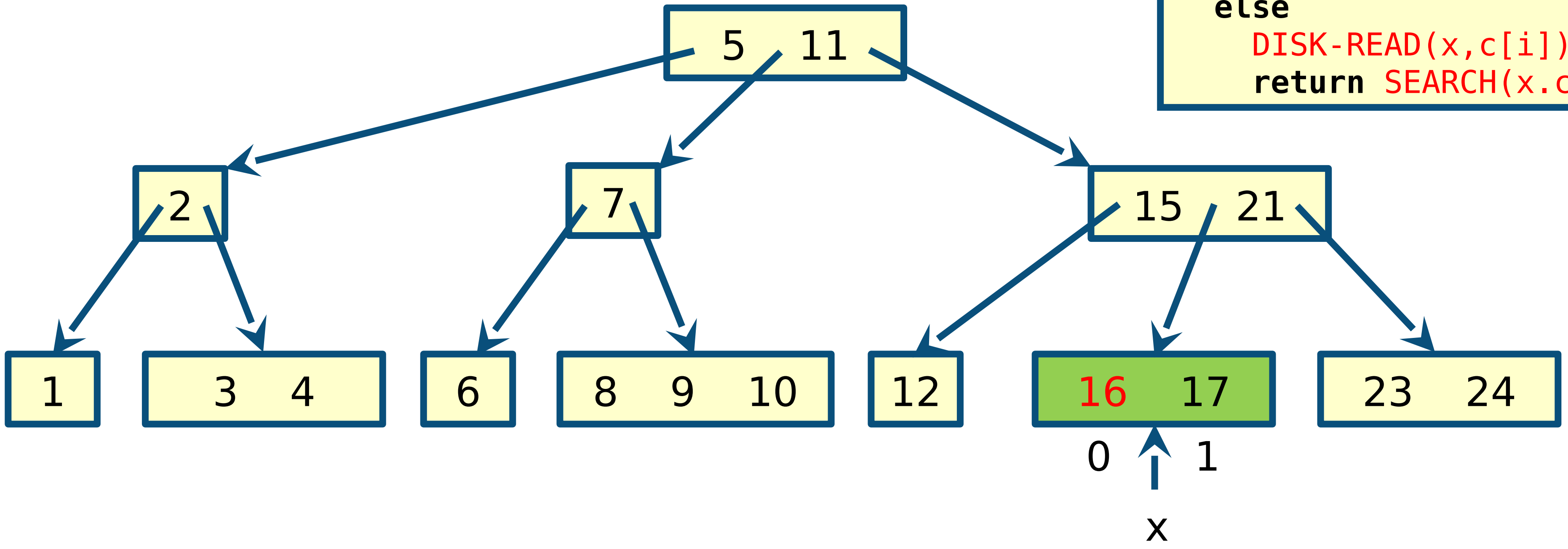
```
SEARCH(x,k)
  i := 0
  while i < x.n and k > x.key[i]
    i := i + 1
  if i < x.n and k = x.key[i]
    return (x,i)
  elseif x.leaf
    return NIL
  else
    DISK-READ(x,c[i])
    return SEARCH(x.c[i],k)
```



Example

- **Find key 16**
  - 16 = x.key[0]
  - Return a pointer to x (the green node) and index 0

```
SEARCH(x,k)
  i := 0
  while i < x.n and k > x.key[i]
    i := i + 1
  if i < x.n and k = x.key[i]
    return (x,i)
  elseif x.leaf
    return NIL
  else
    DISK-READ(x,c[i])
    return SEARCH(x.c[i],k)
```



Creating an empty B-tree

- **Requires O(1) disk operations**
- **O(1) CPU time**

```
CREATE(T)
  x := ALLOCATE-NODE()
  x.leaf := TRUE
  x.n := 0
  DISK-WRITE(x)
  T.root := x
```

Insertion

- **As with BSTs, we search for the leaf position at which to insert a new key**
- **In order not to violate the B-tree properties, a new key is added into an existing leaf node**
- **If a leaf node is full, we call SPLIT-CHILD to split the leaf into two nodes and move the median key up the leaf's parent**
- **If the parent is also full, we must split it before we can insert the new key, and thus we could end up splitting full nodes all the way up the tree**

SPLIT-CHILD

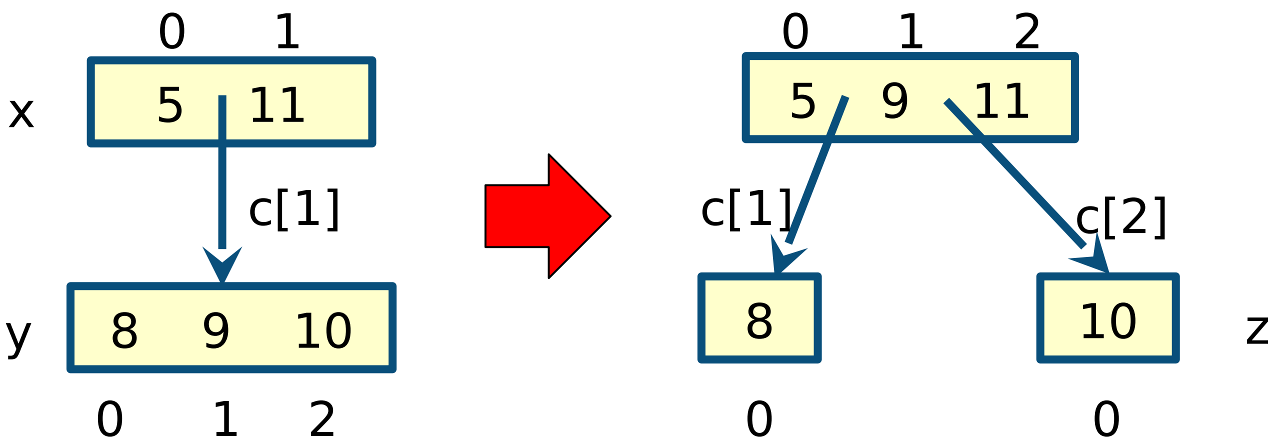
- **Inputs are a nonfull internal node x and an index i such that y = x.c[i] is a full child of x**
  - Both nodes are assumed to be in main memory

- **The procedure splits child y in two and adjusts x so that it has an additional child z**
  - Node y is split about its median key, which is moved up into y's parent node x
  - Keys in y that are greater than the median key are moved into a new node z, which becomes a new child of x

```
SPLIT-CHILD(x,i)
  z := ALLOCATE-NODE()
  y := x.c[i]
  z.leaf := y.leaf
  z.n := t – 1
  for j := 0 to t – 2
    z.key[j] := y.key[j + t]
  if not y.leaf
    for j := 0 to t - 1
      z.c[j] := y.c[j + t]
  y.n := t – 1
  for j := x.n downto i + 1
    x.c[j + 1] := x.c[j]
  x.c[i + 1] := z
  for j := x.n - 1 downto i
    x.key[j + 1] := x.key[j]
  x.key[i] := y.key[t]
  x.n := x.n + 1
  DISK-WRITE(y)
  DISK-WRITE(z)
  DISK-WRITE(x)
```

Example

- **Splitting a node with t = 2**
  - Each node can store at most 2t - 1 = 3 keys
  - Other children are omitted in the representation




```
SPLIT-CHILD(x,i)
  z := ALLOCATE-NODE()
  y := x.c[i]
  z.leaf := y.leaf
  z.n := t – 1
  for j := 0 to t – 2
    z.key[j] := y.key[j + t]
  if not y.leaf
    for j := 0 to t - 1
      z.c[j] := y.c[j + t]
  y.n := t – 1
  for j := x.n downto i + 1
    x.c[j + 1] := x.c[j]
  x.c[i + 1] := z
  for j := x.n - 1 downto i
    x.key[j + 1] := x.key[j]
  x.key[i] := y.key[t]
  x.n := x.n + 1
  DISK-WRITE(y)
  DISK-WRITE(z)
  DISK-WRITE(x)
```

Analysis

- **Total CPU time is O(t)**
- **O(1) disk operations performed**
- **After SPLIT-CHILD is executed, the tree grows in height by one**
  - Splitting is the only means by which the tree grows

```
SPLIT-CHILD(x,i)
  z := ALLOCATE-NODE()
  y := x.c[i]
  z.leaf := y.leaf
  z.n := t – 1
  for j := 0 to t – 2
    z.key[j] := y.key[j + t]
  if not y.leaf
    for j := 0 to t - 1
      z.c[j] := y.c[j + t]
  y.n := t – 1
  for j := x.n downto i + 1
    x.c[j + 1] := x.c[j]
  x.c[i + 1] := z
  for j := x.n - 1 downto i
    x.key[j + 1] := x.key[j]
  x.key[i] := y.key[t]
  x.n := x.n + 1
  DISK-WRITE(y)
  DISK-WRITE(z)
  DISK-WRITE(x)
```

INSERT-NONFULL

- **Procedure to insert key k into node x**
  - x is assumed to be nonfull when the procedure is called

- **There are two cases**

1. If x is a leaf, insert key k in the appropriate position
2. If x is not a leaf node, then insert k into the appropriate leaf node in the subtree rooted at node x

```
INSERT-NONFULL(x,k)
  i := x.n - 1
  if x.leaf
    while i ≥ 0 and k < x.key[i]
      x.key[i + 1] := x.key[i]
      i := i – 1
    x.key[i + 1] := k
    x.n := x.n + 1
    DISK-WRITE(x)
  else while i ≥ 0 and k < x.key[i]
      i := i – 1
    i := i + 1
    DISK-READ(x.c[i])
    if x.c[i].n = 2t – 1
      SPLIT-CHILD(x,i)
      if k > x.key[i]
        i := i + 1
    INSERT-NONFULL(x.c[i],k)
```

INSERT-NONFULL (case 2)

- **The while loop determines the child of x to which the recursion descends**

- **If recursion descends to a full child**
  - Call SPLIT-CHILD
  - Then determine which of the two children is the correct one to descend to

- **One-pass strategy**
  - Every time we encounter a full node, we split it

```
INSERT-NONFULL(x,k)
  i := x.n - 1
  if x.leaf
    while i ≥ 0 and k < x.key[i]
      x.key[i + 1] := x.key[i]
      i := i – 1
    x.key[i + 1] := k
    x.n := x.n + 1
    DISK-WRITE(x)
  else while i ≥ 0 and k < x.key[i]
      i := i – 1
    i := i + 1
    DISK-READ(x.c[i])
    if x.c[i].n = 2t – 1
      SPLIT-CHILD(x,i)
      if k > x.key[i]
        i := i + 1
    INSERT-NONFULL(x.c[i],k)
```

INSERT

- **Procedure to insert key k into a B-tree T**
- **If the root is full, call SPLIT-CHILD before calling INSERT-NONFULL**
- **The total CPU time used is $O(t \log_t n)$**
- **$O(\log_t n)$ disk accesses**

```
INSERT (T,k)
  r := T.root
  if r.n = 2t – 1
    s := ALLOCATE-NODE()
    T.root := s
    s.leaf := FALSE
    s.n := 0
    s.c[0] := r
    SPLIT-CHILD(s,0)
    INSERT-NONFULL(s,k)
   else
    INSERT-NONFULL(r,k)
```

Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most $2t - 1 = 3$ keys

Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys

Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys

$\boxed{5 \quad 7}$

Example

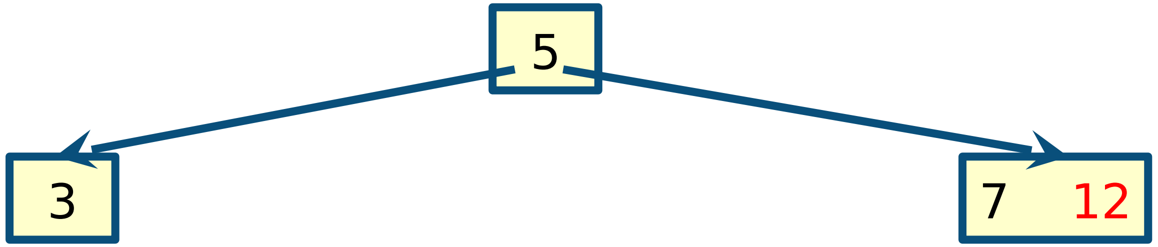
- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most $2t - 1 = 3$ keys

3	5	7

full node

Example

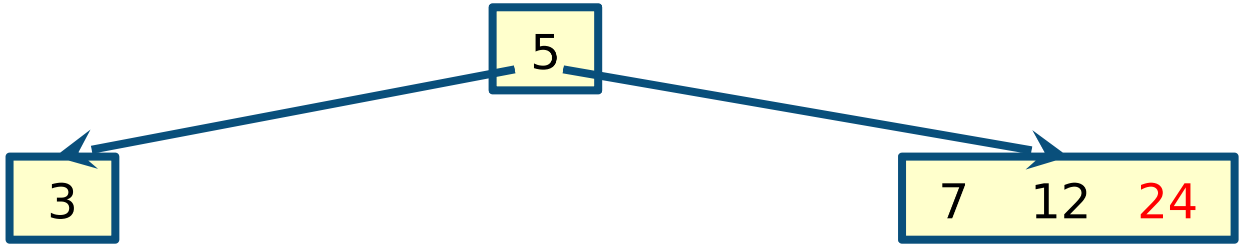
- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



SPLIT

Example

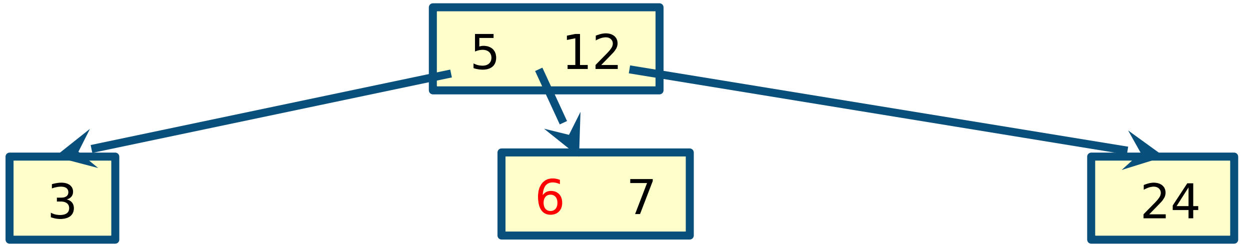
- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



full node

Example

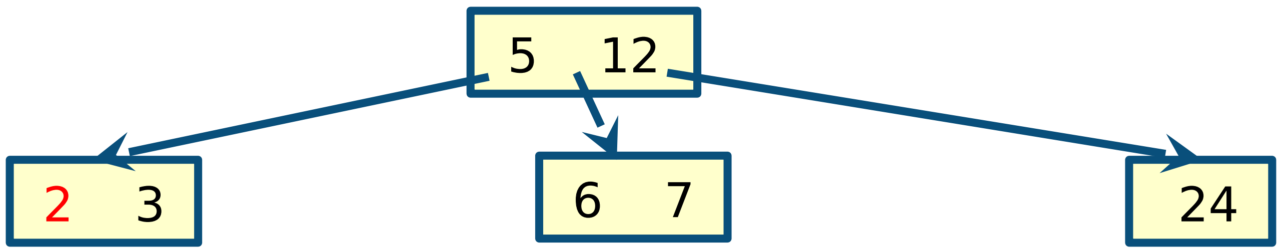
- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



SPLIT

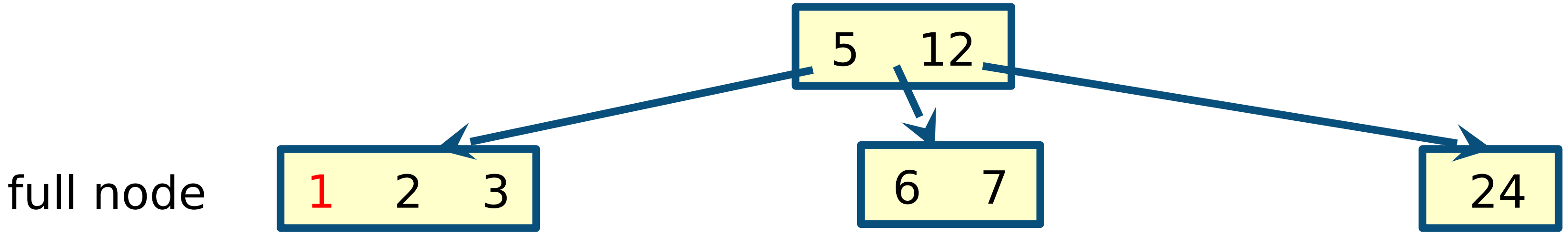
Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



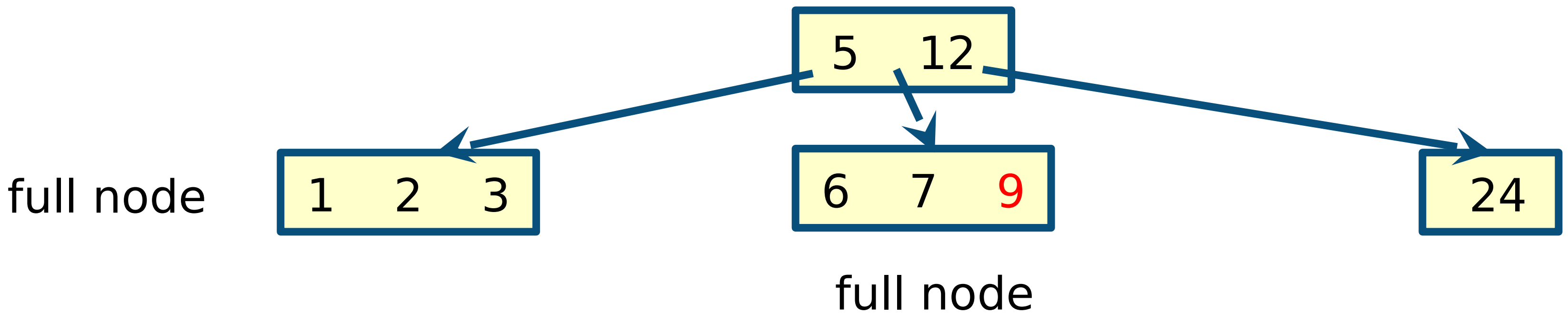
Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



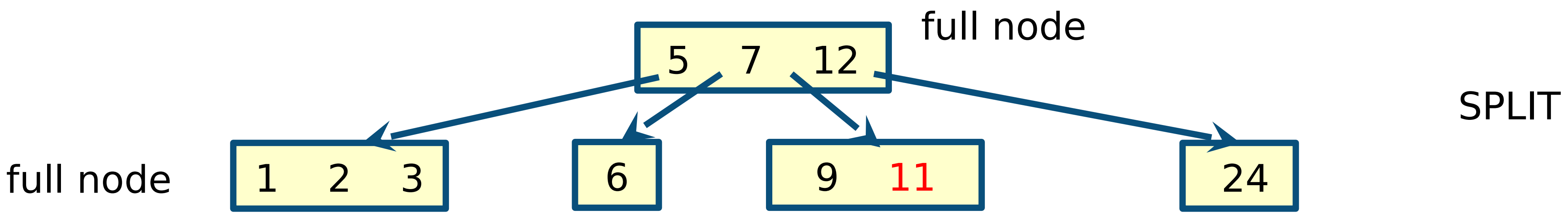
Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



Example

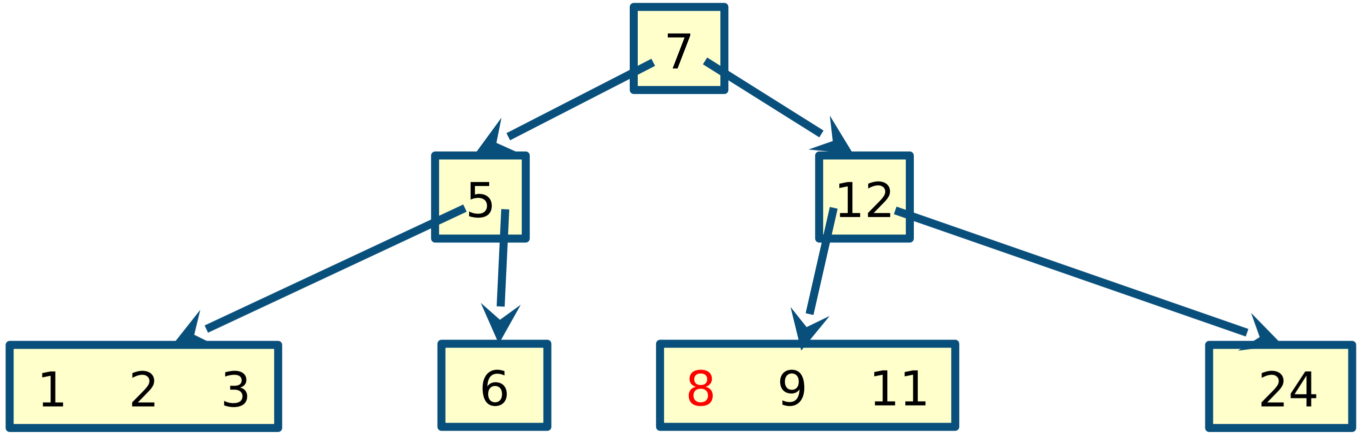
- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys

full node

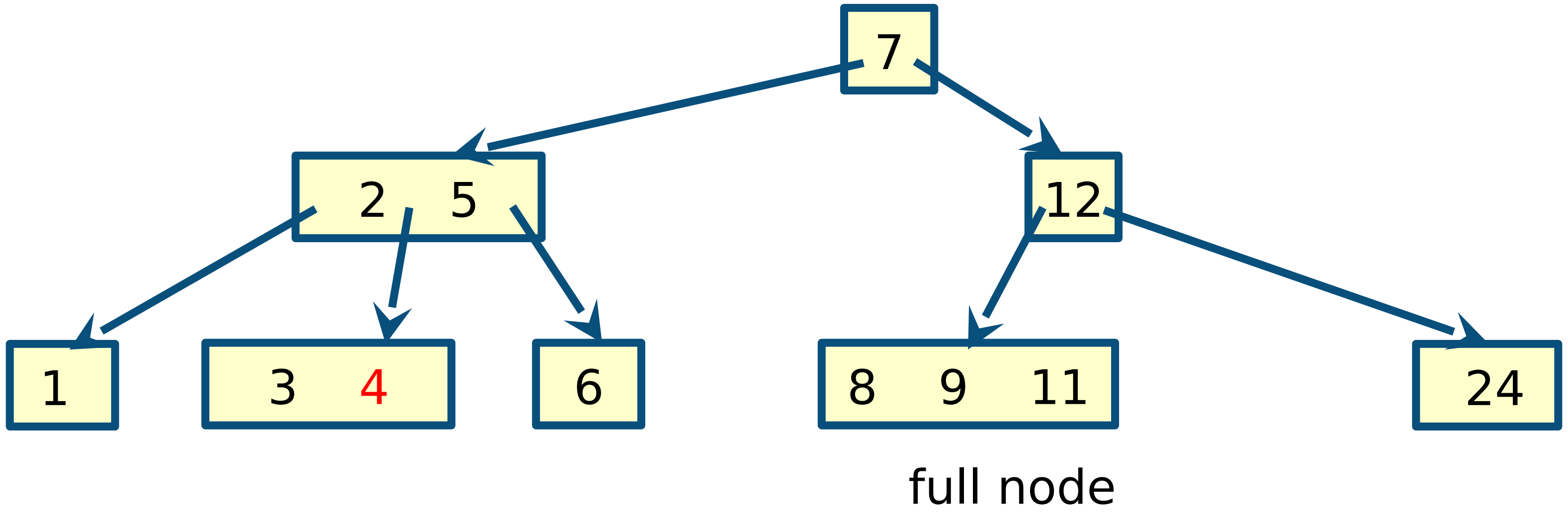


full node

SPLIT

Example

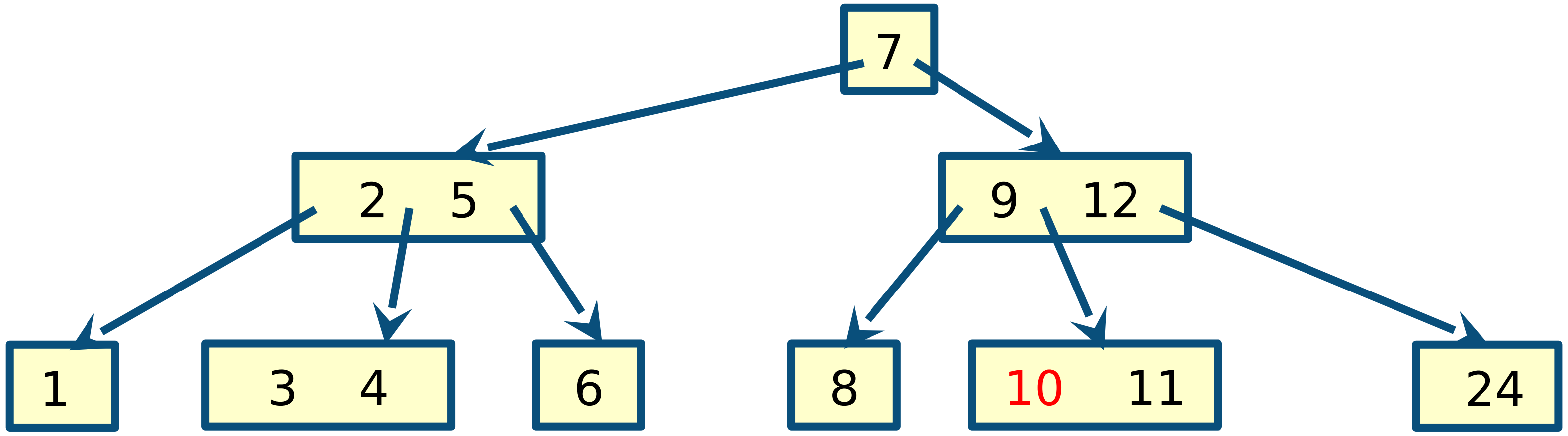
- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



SPLIT

Example

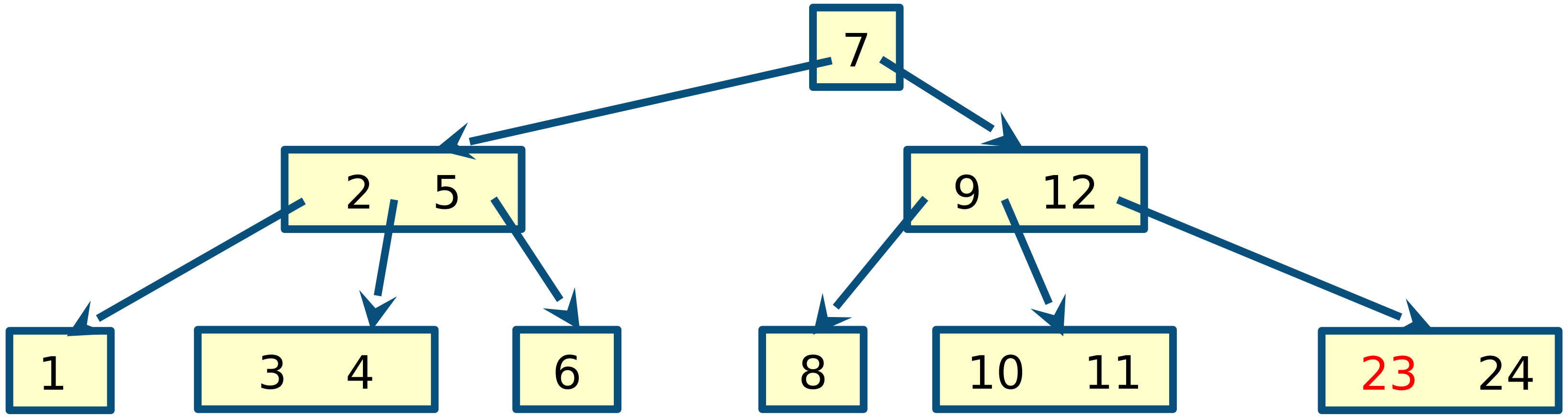
- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



SPLIT

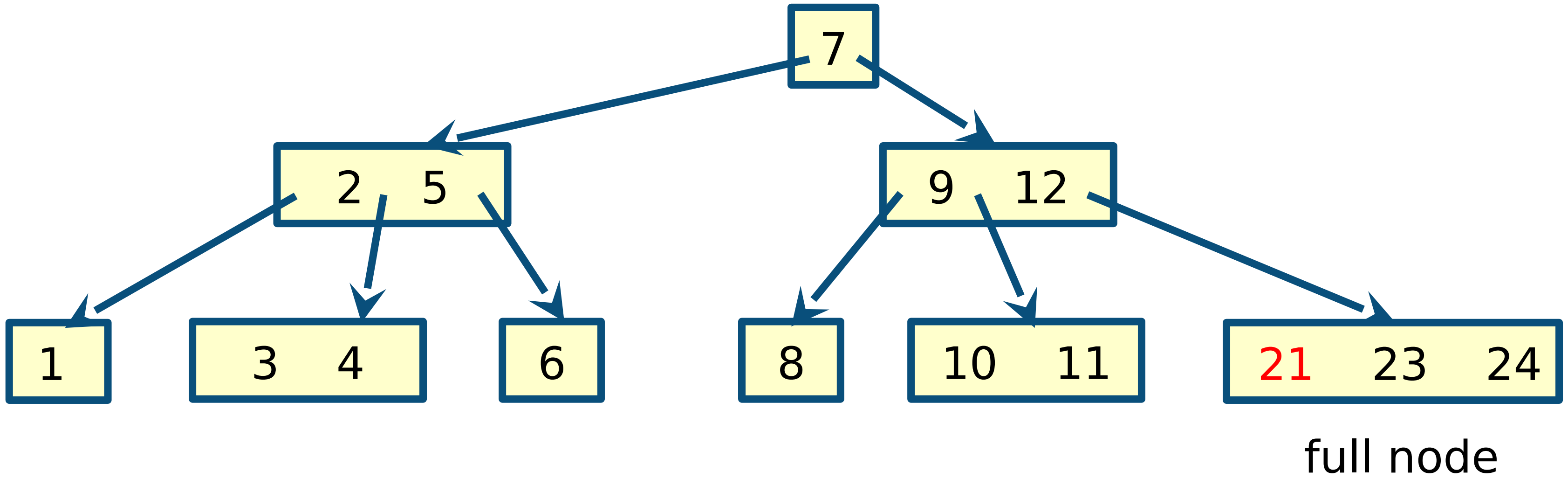
Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



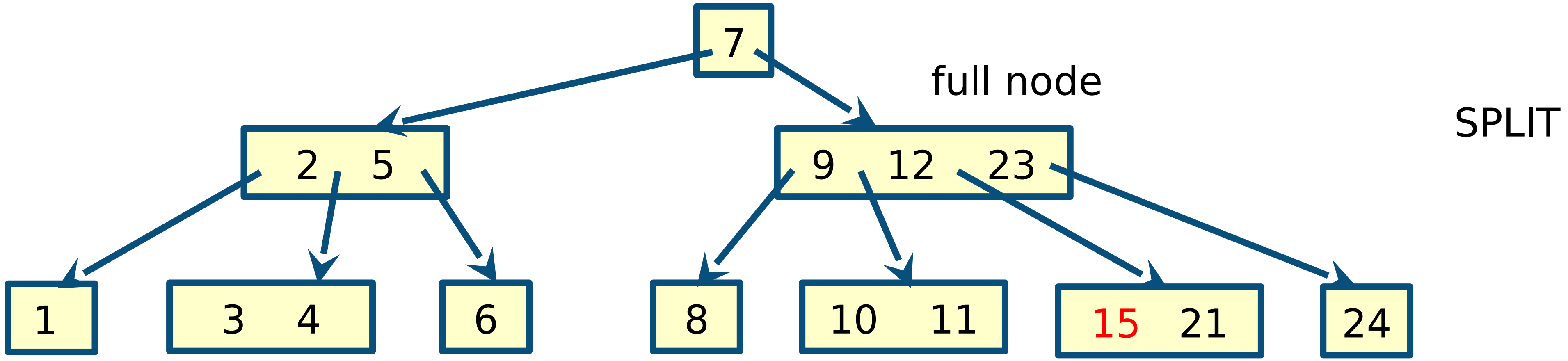
Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



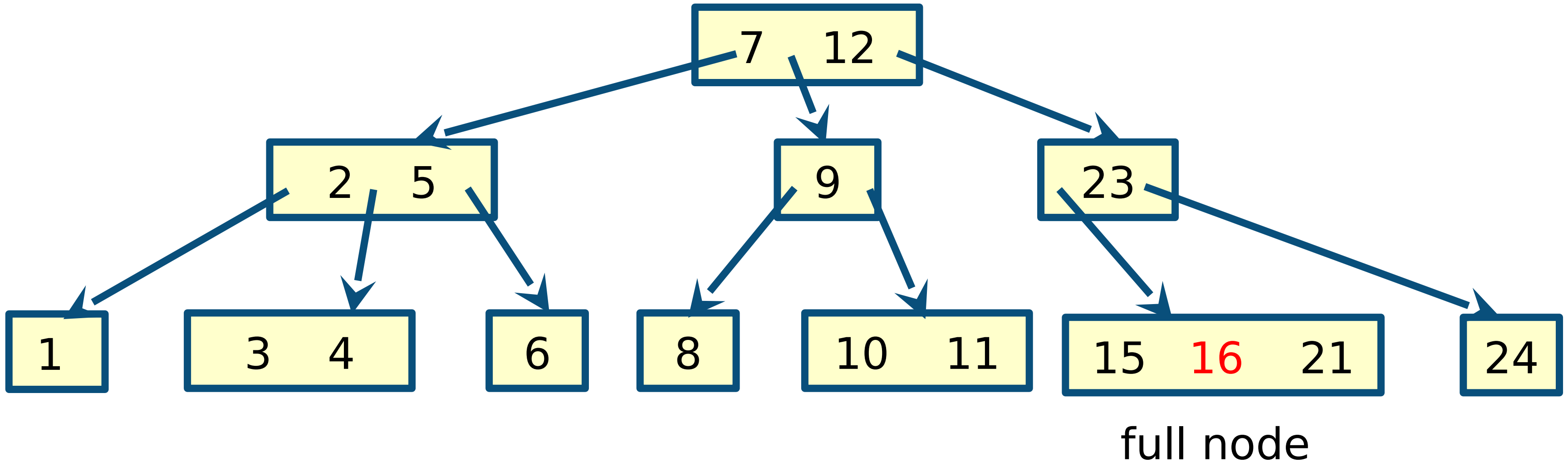
Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



Example

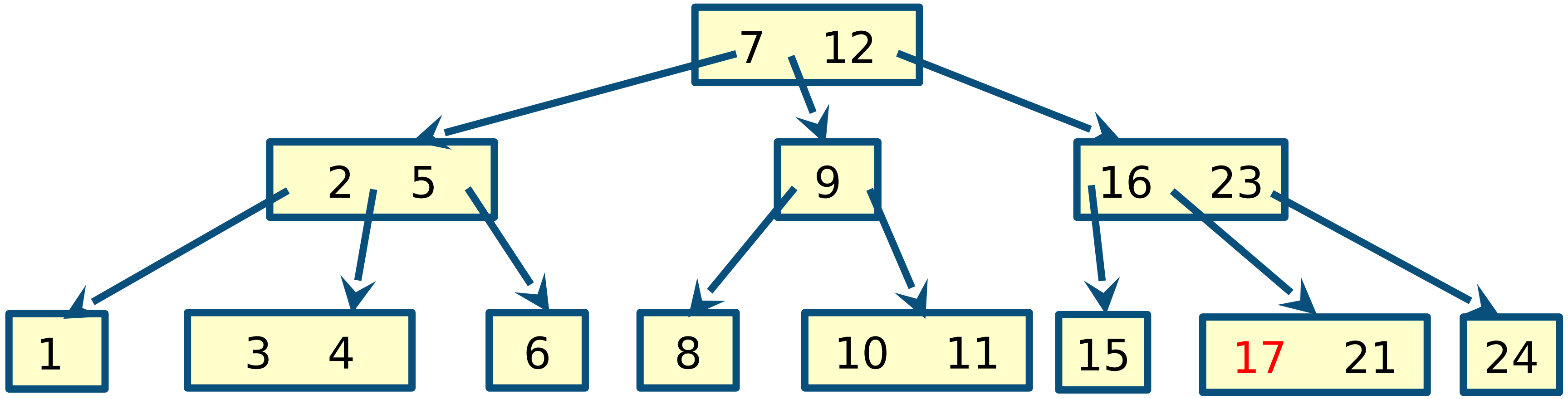
- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



SPLIT

Example

- **Add the sequence of keys given below to an empty B-tree with t = 2**
  - 5,7,3,12,24,6,2,1,9,11,8,4,10,23,21,15,16,17
  - Each node can store at most 2t - 1 = 3 keys



SPLIT

Variants

- **2-3 trees are B-trees in which every internal node has either two or three children**
  - 2-3-4 trees can have up to 4 children (see slide 6)

- **B+trees store all the satellite information in the leaves and stores only keys and child pointers in the internal nodes**
  - To maximise their branching factor

- **B* trees balance more neighbouring internal nodes to keep them more densely packed**
  - This variant ensures non-root nodes are at least 2/3 full instead of 1/2

Applications

- **Example databases using B-trees**
  - MySQL uses both B-Tree and B+Tree indices
  - Oracle Database uses a B-tree index
  - Microsoft’s SQL Server uses a B+Tree index

- **Example filesystems based on B-trees**
  - Apple's HFS+
  - Microsoft's NTFS
  - Linux Ext4
  - Reiser4 uses B*-trees
  - DragonFly BSD's HAMMER uses B+trees

Summary

- **Definition**
- **Motivation**
- **Properties**
- **Operations**
  - Search
  - Insertions
- **Variants**
- **Applications**