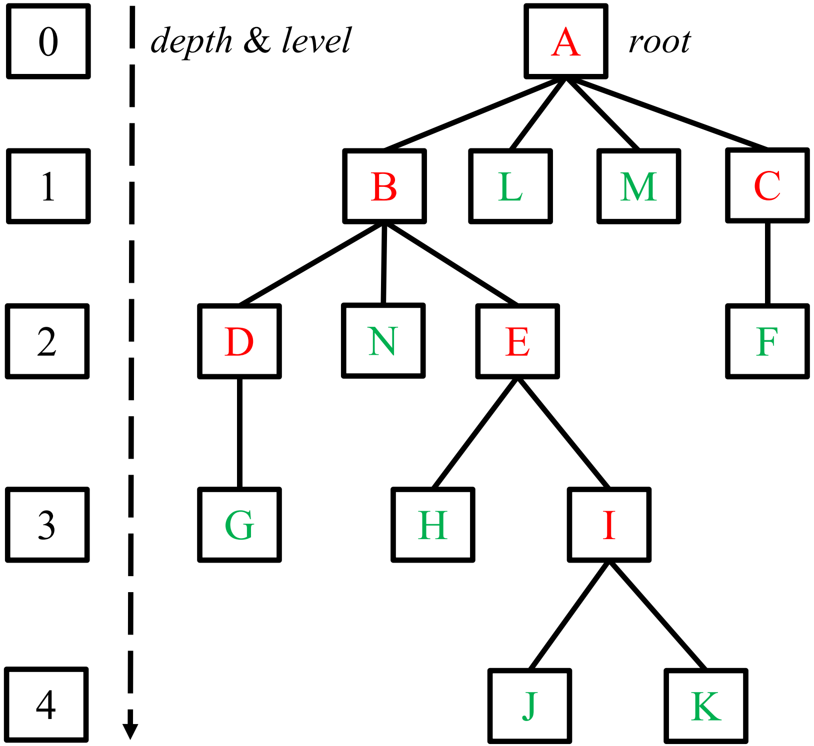
General Tree & Indexing

LI Hao 李颢, Assoc. Prof. SPEIT & Dept. Automation of SEIEE

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

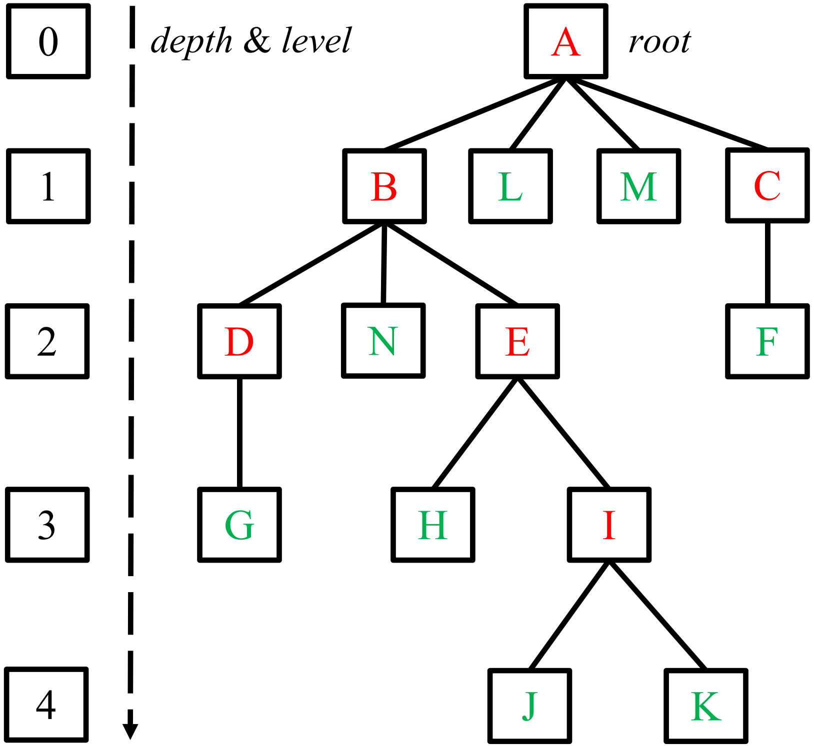
General Tree

- **General tree**
  - *node & edge*
  - *root*
  - arbitrary number of *subtrees*
  - *parent* & *children* (not limited to 2)
  - *ancestor & descendant*
  - *path & length*
    - B-E-I, A-C-F : two paths of length 2
    - A-B-E-I-J: a path of length 4
  - *depth* (cardinal) & *level* (ordinal)
  - *height* (largest depth+1)
  - *leaf node* & *internal node*



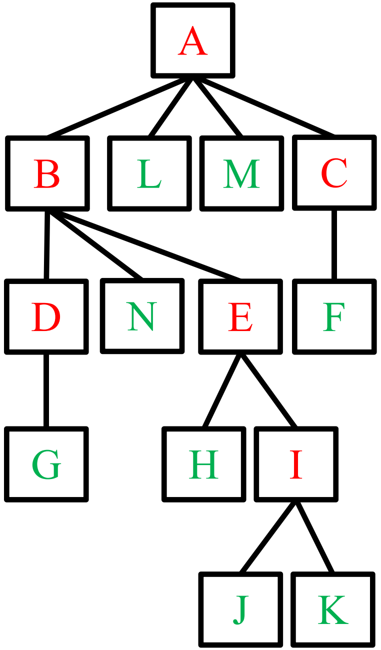
General Tree

- **General tree**
  - *node & edge*
  - *root*
  - arbitrary number of *subtrees*
  - *parent* & *children* (not limited to 2)
  - *out degree* - number of children
  - *left-most child* - arranged from left to right
  - *ancestor & descendant*
  - *path & length*
  - *depth* (cardinal) & *level* (ordinal)
  - *height* (largest depth+1)
  - *leaf node* & *internal node*



General Tree

- **General tree**
  - *left-child/right-sibling* implementation

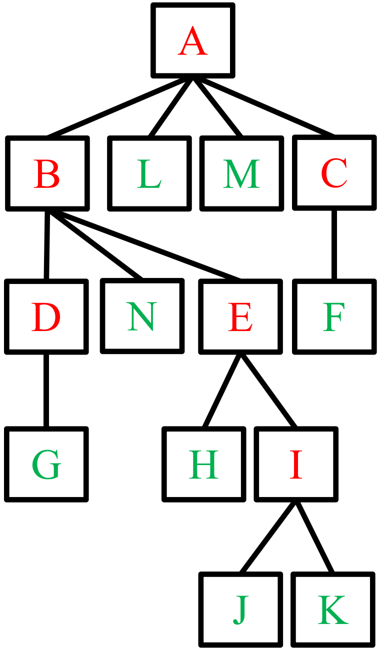



```
template <class T> class GTNode{ // GT (general tree) node abstract class
private:T e; // node's element
        GTNode *cL,*sR,*p; // node's leftmost child, right sibling, parent
public: GTNode(){cL=sR=p=NULL;} ~GTNode(){}
        GTNode(const T& ei,GTNode* cLi=NULL,GTNode* sRi=NULL,GTNode* pi=NULL){
                e=ei;cL=cLi;sR=sRi;p=pi;}
        const T& getE() const{return e;} void setE(const T& ei){e=ei;}
        inline GTNode* getC() const{return cL;} void setC(GTNode* g){cL=g;}
        inline GTNode* getS() const{return sR;} void setS(GTNode* g){sR=g;}
        inline GTNode* getP() const{return p;} void setP(GTNode* g){p=g;}
        bool isLeaf(){return cL==NULL;}
};

inline void indent(int L){while(L--) std::cout<<"|...";}
// preorder traversal of GT
template <class T> void showSubGT(GTNode<T>* g,int L){
        if(g==NULL) return;indent(L);std::cout<<g->getE()<<'\n';
        g=g->getC();while(g!=NULL){showSubGT(g,L+1);g=g->getS();}}
template <class T> void showGT(GTNode<T>* g){
        while(g->getP()!=NULL) g=g->getP(); showSubGT(g,0);}
```

General Tree

- **General tree**
  - *left-child/right-sibling* implementation



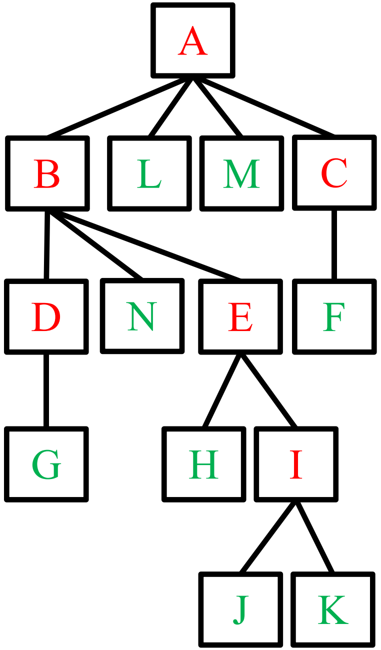

```
sub-GT of B =>
B
|...D
|...|...G
|...N
|...E
|...|...H
|...|...I
|...|...|...J
|...|...|...K
```

```
GT to which B belongs =>
A
|...B
|...|...D
|...|...|...G
|...|...N
|...|...E
|...|...|...H
|...|...|...I
|...|...|...|...J
|...|...|...|...K
|...L
|...M
|...C
|...|...F
```

```
GTNode<char> *A=new GTNode<char>('A');GTNode<char> *B=new GTNode<char>('B');GTNode<char> *C=new GTNode<char>('C');
GTNode<char> *D=new GTNode<char>('D');GTNode<char> *E=new GTNode<char>('E');GTNode<char> *F=new GTNode<char>('F');
GTNode<char> *G=new GTNode<char>('G');GTNode<char> *H=new GTNode<char>('H');GTNode<char> *I=new GTNode<char>('I');
GTNode<char> *J=new GTNode<char>('J');GTNode<char> *K=new GTNode<char>('K');GTNode<char> *L=new GTNode<char>('L');
GTNode<char> *M=new GTNode<char>('M');GTNode<char> *N=new GTNode<char>('N');
A->setC(B);B->setP(A);B->setS(L);B->setC(D);C->setP(A);C->setC(F);D->setP(B);D->setS(N);D->setC(G);
E->setP(B);E->setC(H);F->setP(C);G->setP(D);H->setP(E);H->setS(I);I->setP(E);I->setC(J);J->setP(I);
J->setS(K);K->setP(I);L->setP(A);L->setS(M);M->setP(A);M->setS(C);N->setP(B);N->setS(E);
cout<<"sub-GT of B => \n";showSubGT<char>(B,0);cout<<"GT to which B belongs => \n";showGT<char>(B);
```

General Tree

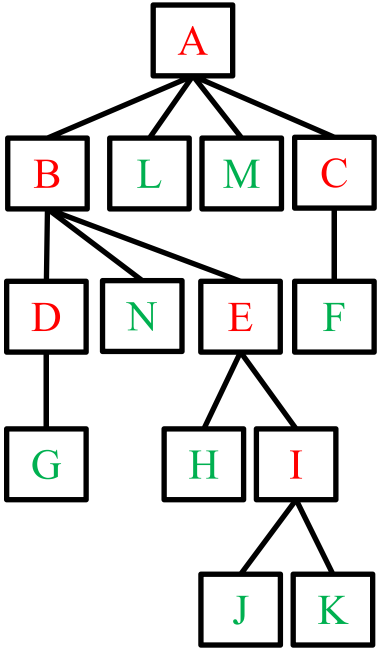
- **General tree**
  - *left-child/right-sibling* implementation




```
template <class T> class LGT{ // linked general tree abstract class
private:GTNode<T>* r; // GT root
        inline void indent(int L){while(L--) std::cout<<"|---";}
        void SIn(GTNode<T>* g,int L);
        void clearIn(GTNode<T>* g);
public: LGT(const T& ei){r=new GTNode<T>(ei);} ~LGT(){}
        inline GTNode<T>* getR() const{return r;}
        void setC(LGT* gt){r->setC(gt->getR());delete gt;} // merge 'gt' as left-most child & deallocate it
        void setS(LGT* gt){r->setS(gt->getR());delete gt;} // merge 'gt' as right sibling & deallocate it
        // setC() & secS() deallocate gt's LGT space but not gt's associated GTNodes space
        void clear(); // deallocate root's associated GTNodes space
        void S(); // preorder traversal of GT
};
template <class T> void LGT<T>::SIn(GTNode<T>* g,int L){if(g==NULL) return;
        while(g!=NULL){indent(L);std::cout<<g->getE()<<'\n';SIn(g->getC(),L+1);g=g->getS();}}
// default arguments are only allowed in function declarations (so omitted in S() definition here)
template <class T> void LGT<T>::S(){SIn(r,0);}
template <class T> void LGT<T>::clearIn(GTNode<T>* g){if(g==NULL) return;
        GTNode<T>* t=g->getC();delete g;g=t;while(g!=NULL){t=g->getS();clearIn(g);g=t;}}
template <class T> void LGT<T>::clear(){clearIn(r);r=NULL;}
```

General Tree

- **General tree**
  - *left-child/right-sibling* implementation

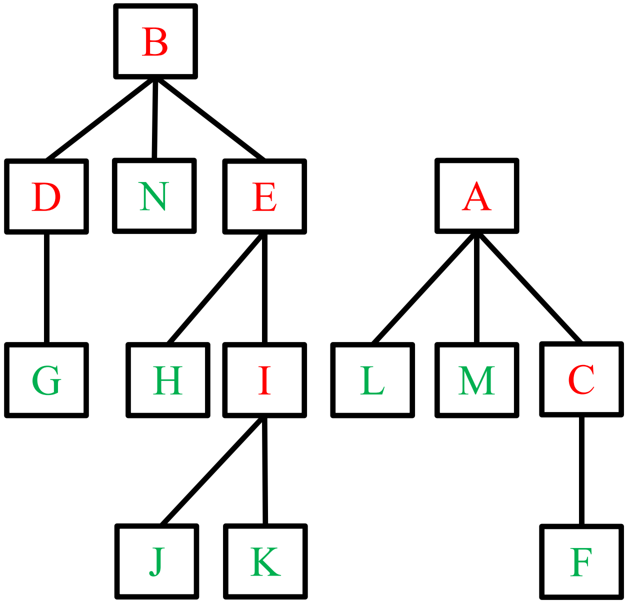



```
GT of A =>
A
|---B
|---|---D
|---|---|---G
|---|---N
|---|---E
|---|---|---H
|---|---|---I
|---|---|---|---J
|---|---|---|---K
|---L
|---M
|---C
|---|---F
GT of A after clearing =>
```

```
LGT<char> *gtA=new LGT<char>('A');LGT<char> *gtB=new LGT<char>('B');LGT<char> *gtC=new LGT<char>('C');
LGT<char> *gtD=new LGT<char>('D');LGT<char> *gtE=new LGT<char>('E');LGT<char> *gtF=new LGT<char>('F');
LGT<char> *gtG=new LGT<char>('G');LGT<char> *gtH=new LGT<char>('H');LGT<char> *gtI=new LGT<char>('I');
LGT<char> *gtJ=new LGT<char>('J');LGT<char> *gtK=new LGT<char>('K');LGT<char> *gtL=new LGT<char>('L');
LGT<char> *gtM=new LGT<char>('M');LGT<char> *gtN=new LGT<char>('N');
gtJ->setS(gtK);gtI->setC(gtJ);gtH->setS(gtI);gtE->setC(gtH);gtN->setS(gtE);gtD->setC(gtG);gtD->setS(gtN);
gtB->setC(gtD);gtC->setC(gtF);gtM->setS(gtC);gtL->setS(gtM);gtB->setS(gtL);gtA->setC(gtB);
cout<<"GT of A =>\n";gtA->S();gtA->clear();cout<<"GT of A after clearing =>\n";gtA->S();
```

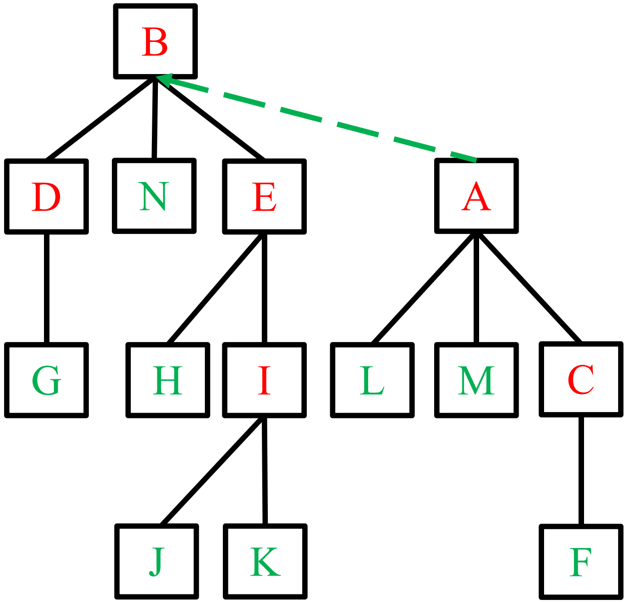
General Tree

- **General tree**
  - *parent pointer* implementation
    - determine if two nodes are in the same tree
      - e.g. G->D->B, K->I->E->B, so G & K in same tree
      - e.g. H->E->B, L->A, so H & L not in same tree
    - merge two trees



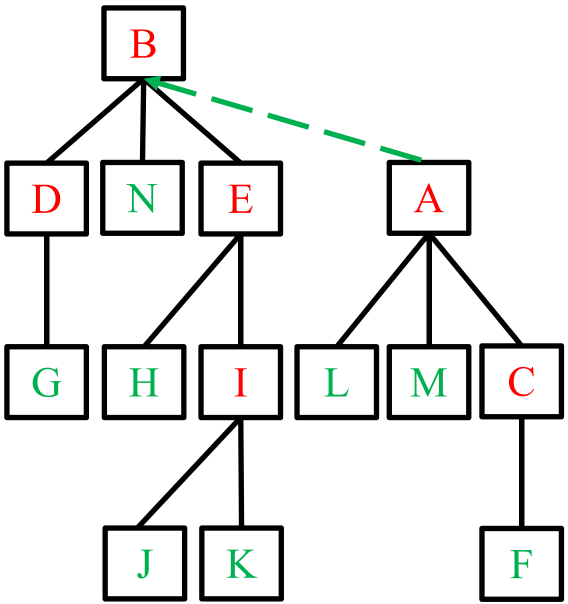
General Tree

- **General tree**
  - *parent pointer* implementation
    - determine if two nodes are in the same tree
      - e.g. G->D->B, K->I->E->B, so G & K in same tree
      - e.g. H->E->B, L->A, so H & L not in same tree
    - merge two trees
      - set a tree's root as parent of the other's root



General Tree

- **General tree**
  - *parent pointer* implementation

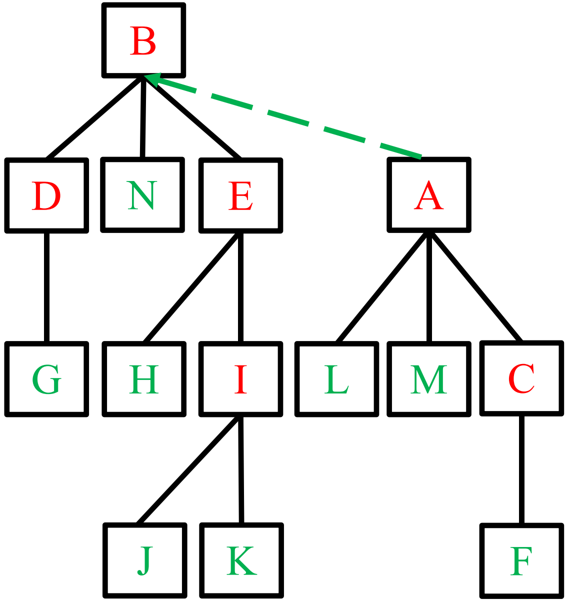



```
template <class T> class PPNode{ // parent pointer GT node abstract class
private:T e;int n; // node's element; number of nodes of the node's tree
        PPNode *p; // node's parent
public: PPNode(){p=NULL;n=1;} ~PPNode(){} PPNode(const T& ei){e=ei;p=NULL;n=1;}
        const T& getE() const{return e;} void setE(const T& ei){e=ei;}
        int getN() const{return n;} void setN(int ni){n=ni;}
        inline PPNode* getP() const{return p;} void setP(PPNode* g){p=g;}
        static PPNode* find(PPNode* g){while(g->getP()!=NULL) g=g->getP();return g;}
        static void ppunion(PPNode* a,PPNode* b){a=find(a);b=find(b);if(a==b) return;
                if(a->getN()<=b->getN()){a->setP(b);b->setN(b->getN()+a->getN());}
                else{b->setP(a);a->setN(a->getN()+b->getN());}}
};
// ostream overloading, so that 'cout<<{PPNode<T> object}' can have meaning
template <typename T> std::ostream& operator<<(std::ostream& out,PPNode<T>* b){
        out<<b->getE()<<':'<<b->getN(); return out;}
```

```
PPNode<char> *p1,*p2;PPNode<char>* pp[26]={new PPNode<char>('A'),
new PPNode<char>('B'), new PPNode<char>('C'), new PPNode<char>('D'), new PPNode<char>('E'), new PPNode<char>('F'),
new PPNode<char>('G'), new PPNode<char>('H'), new PPNode<char>('I'), new PPNode<char>('J'), new PPNode<char>('K'),
new PPNode<char>('L'), new PPNode<char>('M'), new PPNode<char>('N'), new PPNode<char>('O'), new PPNode<char>('P'),
new PPNode<char>('Q'), new PPNode<char>('R'), new PPNode<char>('S'), new PPNode<char>('T'), new PPNode<char>('U'),
new PPNode<char>('V'), new PPNode<char>('W'), new PPNode<char>('X'), new PPNode<char>('Y'), new PPNode<char>('Z')};
while(true){int i=rand()%26,j=rand()%26;p1=PPNode<char>::find(pp[i]);p2=PPNode<char>::find(pp[j]);
        cout<<"merge "<<pp[i]<<" & "<<pp[j]<<" : "<<pp[i]<<" => "<<p1<<" | "<<pp[j]<<" => "<<p2<<" : ";
        if(p1==p2){cout<<p1<<"=="<<p2<<" unnecessary to merge!\n";}
        else{PPNode<char>::ppunion(p1,p2);
                if(p1->getN()<p2->getN()){cout<<" merge into "<<p2<<endl;if(26==p2->getN()) break;}
                else{cout<<" merge into "<<p1<<endl;if(26==p1->getN()) break;}}}
```

General Tree

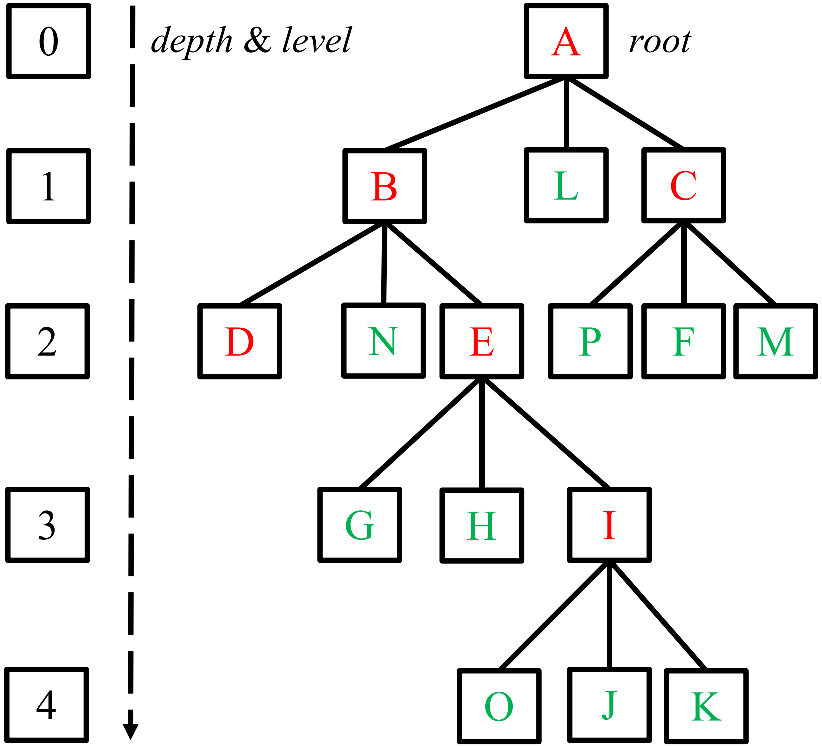
- **General tree**
  - *parent pointer* implementation




```
merge N:1 & W:1 : N:1 => N:1 | W:1 => W:1 :  merge into W:2
merge L:1 & R:1 : L:1 => L:1 | R:1 => R:1 :  merge into R:2
merge B:1 & B:1 : B:1 => B:1 | B:1 => B:1 : B:1==B:1 unnecessary to merge!
merge M:1 & Q:1 : M:1 => M:1 | Q:1 => Q:1 :  merge into Q:2
merge B:1 & H:1 : B:1 => B:1 | H:1 => H:1 :  merge into H:2
merge C:1 & D:1 : C:1 => C:1 | D:1 => D:1 :  merge into D:2
merge A:1 & R:2 : A:1 => A:1 | R:2 => R:2 :  merge into R:3
merge Z:1 & O:1 : Z:1 => Z:1 | O:1 => O:1 :  merge into O:2
merge W:2 & K:1 : W:2 => W:2 | K:1 => K:1 :  merge into W:3
merge K:1 & Y:1 : K:1 => W:3 | Y:1 => Y:1 :  merge into W:4
merge H:2 & I:1 : H:2 => H:2 | I:1 => I:1 :  merge into H:3
merge D:2 & D:2 : D:2 => D:2 | D:2 => D:2 : D:2==D:2 unnecessary to merge!
merge Q:2 & S:1 : Q:2 => Q:2 | S:1 => S:1 :  merge into Q:3
merge C:1 & D:2 : C:1 => D:2 | D:2 => D:2 : D:2==D:2 unnecessary to merge!
merge X:1 & R:3 : X:1 => X:1 | R:3 => R:3 :  merge into R:4
merge J:1 & M:1 : J:1 => J:1 | M:1 => Q:3 :  merge into Q:4
merge O:2 & W:4 : O:2 => O:2 | W:4 => W:4 :  merge into W:6
merge F:1 & R:4 : F:1 => F:1 | R:4 => R:4 :  merge into R:5
merge X:1 & S:1 : X:1 => R:5 | S:1 => Q:4 :  merge into R:9
merge J:1 & Y:1 : J:1 => R:9 | Y:1 => W:6 :  merge into R:15
merge B:1 & L:1 : B:1 => H:3 | L:1 => R:15 :  merge into R:18
merge D:2 & B:1 : D:2 => D:2 | B:1 => R:18 :  merge into R:20
merge E:1 & F:1 : E:1 => E:1 | F:1 => R:20 :  merge into R:21
merge S:1 & A:1 : S:1 => R:21 | A:1 => R:21 : R:21==R:21 unnecessary to merge!
merge R:21 & C:1 : R:21 => R:21 | C:1 => R:21 : R:21==R:21 unnecessary to merge!
merge B:1 & Y:1 : B:1 => R:21 | Y:1 => R:21 : R:21==R:21 unnecessary to merge!
merge N:1 & E:1 : N:1 => R:21 | E:1 => R:21 : R:21==R:21 unnecessary to merge!
merge C:1 & D:2 : C:1 => R:21 | D:2 => R:21 : R:21==R:21 unnecessary to merge!
merge Y:1 & G:1 : Y:1 => R:21 | G:1 => G:1 :  merge into R:22
merge G:1 & X:1 : G:1 => R:22 | X:1 => R:22 : R:22==R:22 unnecessary to merge!
merge X:1 & P:1 : X:1 => R:22 | P:1 => P:1 :  merge into R:23
merge K:1 & L:1 : K:1 => R:23 | L:1 => R:23 : R:23==R:23 unnecessary to merge!
```

General Tree

- **K-ary tree**
  - *node & edge*
  - *root*
  - 0 or K *subtrees* e.g. binary (2-ary) tree
  - *parent* & *children* (0 or K)
  - *ancestor & descendant*
  - *path & length*
    - B-E-I, A-C-F : two paths of length 2
    - A-B-E-I-J: a path of length 4
  - *depth* (cardinal) & *level* (ordinal)
  - *height* (largest depth+1)
  - *leaf node* & *internal node*



THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Indexing

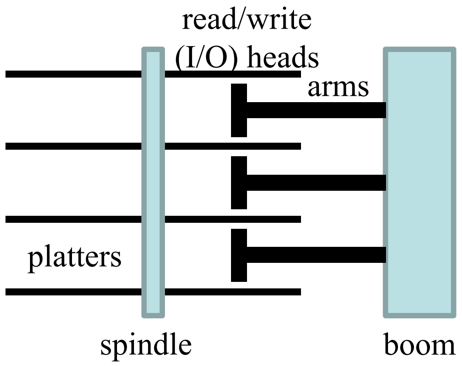
REVIEW

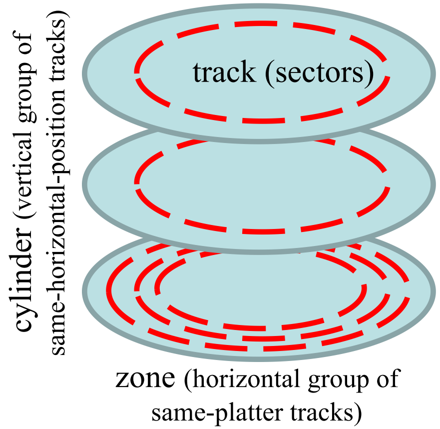
- **Primary (main) memory vs. secondary (peripheral) storage**
  - primary or main memory
    - e.g. **random access memory** (RAM), registers, cache, video memories
  - secondary or peripheral storage
    - e.g. **hard disk drives**, solid state drives, removable USB drives, CDs, DVDs
  - memory & storage access speed
    - RAM access speed is $10^5 \sim 10^6$ faster than disk drive access speed
- **Disk-based applications - minimize the number of disk accesses**
  - organize file structures
  - save information previously retrieved

Indexing

REVIEW

- **Disk drive access**
  - disk drive architecture
  - disk drive acess
    - seek
    - preparatory rotation
    - data transfer
  - disk drive access time (e.g.)
    - disk drive (20G) = 10 platters ; platter = 16384 tracks
    - track (128K) = 32 clusters = 256 sectors ; sector = 0.5K ; cluster = 4K
    - track-to-track seek = 2.0ms ; random seek = 8.0ms ; rotation = 8.33ms (7200 RPM)
    - **how long will it take to read a file of 1M** (1024K = 8 tracks = 256 clusters)?
    - if the file fills 8 adjacent tracks : 122ms
    - if the file fills 256 clusters spread randomly across the disk
      - 256x[8.0+8.33x(0.5+8/256)]=256x(8.0+4.43)=3182ms





Indexing

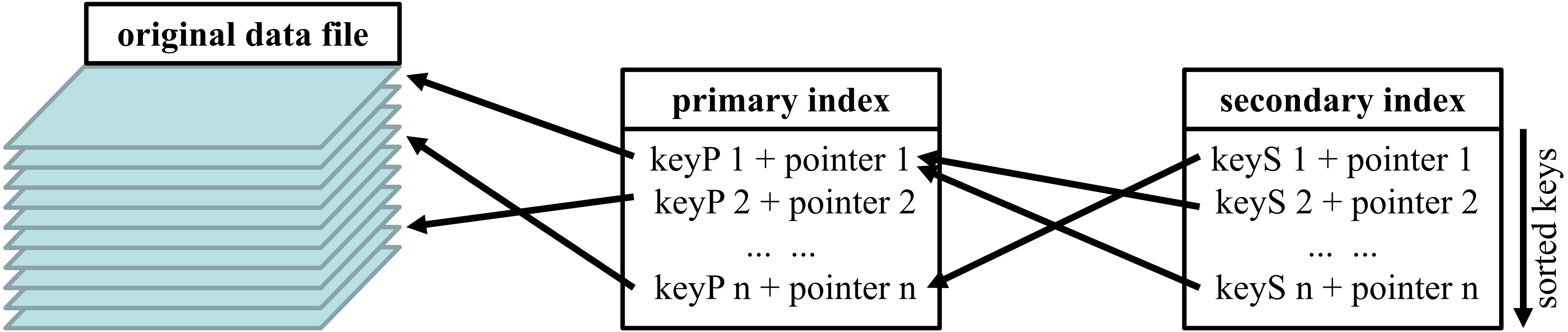
REVIEW

- **Disk drive access**
  - disk drive access time (e.g.)
    - disk drive (20G) = 10 platters ; platter = 16384 tracks
    - track (128K) = 32 clusters = 256 sectors ; sector = 0.5K ; cluster = 4K
    - track-to-track seek = 2.0ms ; random seek = 8.0ms ; rotation = 8.33ms (7200 RPM)
    - access a track : 8.0+8.33x(0.5+1)=20.50ms
    - access a sector : 8.0+8.33x(0.5+1/256)=12.20ms (saves almost a half of track-access time)
    - access a byte : 8.0+8.33x(0.5+1/128K)=12.17ms (save almost none of sector-access time)
  - **automatic reading/writing of an entire sector**
  - *buffering* a.k.a. *caching*
    - take/send additional information from/to disk to satisfy (potential) future requests
    - most operating systems maintain at least two buffers, one for input, one for output

Indexing

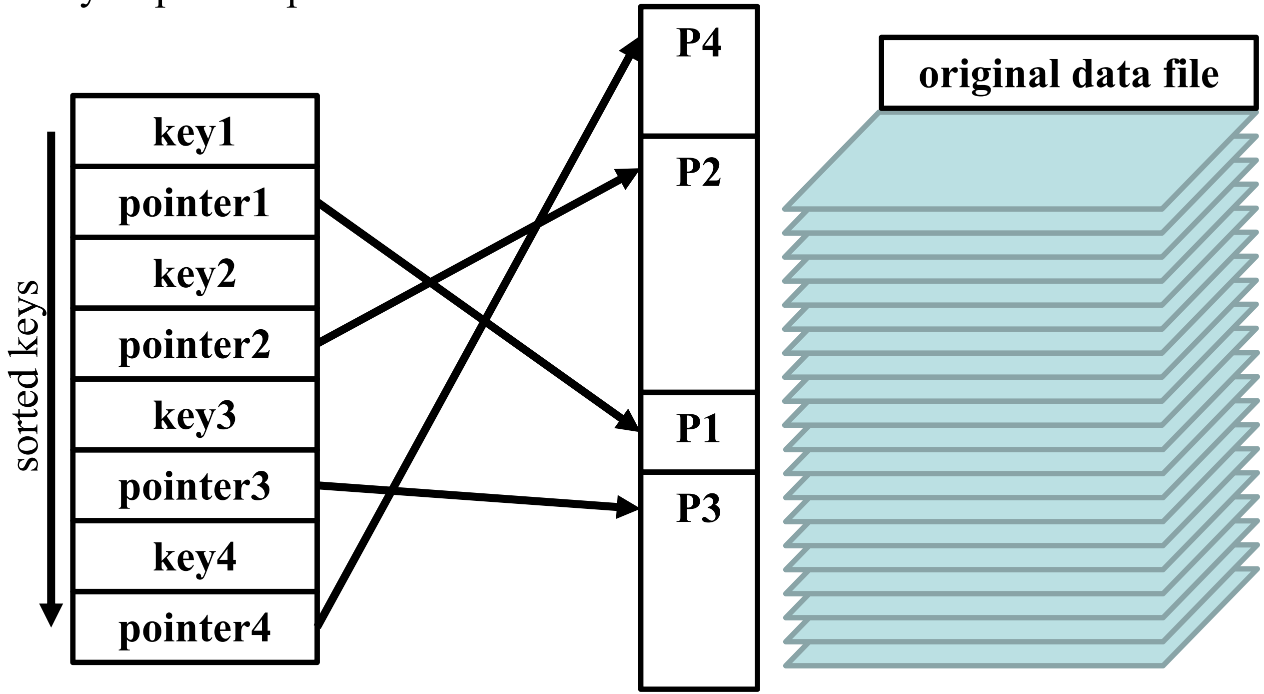
REVIEW

- **Index file - disk-based applications**
  - key sort & pointer methodology
  - **primary (key) index**
    - each record normally has a *unique identifier* called the primary key
    - relate each primary key value with a *pointer* to the actual record on disk
  - **secondary (key) index**
    - normally refer to the *primary index* instead of directly to the actual record on disk



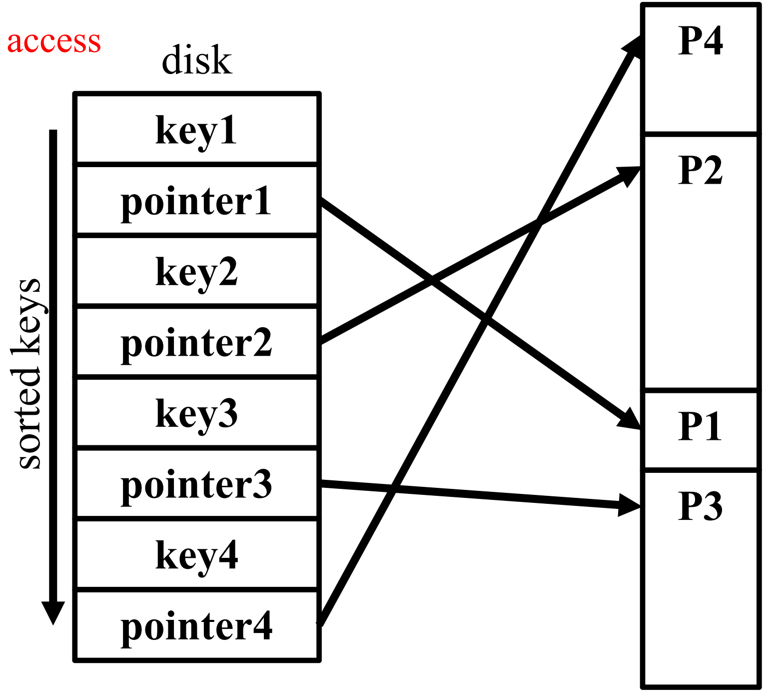
Indexing

- **Linear indexing**
  - fixed-length key & pointer pairs



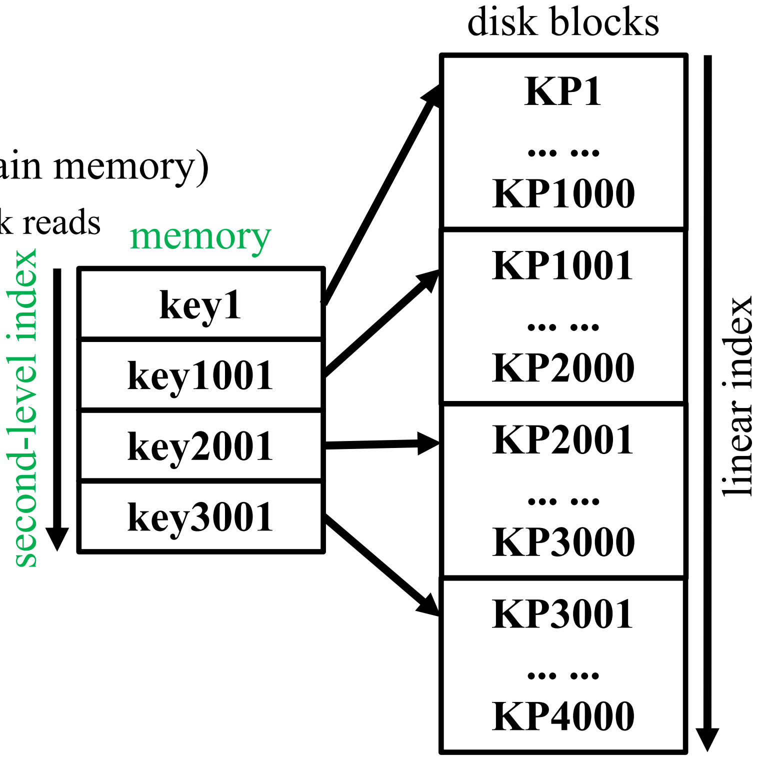
Indexing

- **Linear indexing**
  - fixed-length key & pointer pairs
    - searching involves a lot of inefficient disk access
    - automatic I/O of each entire block



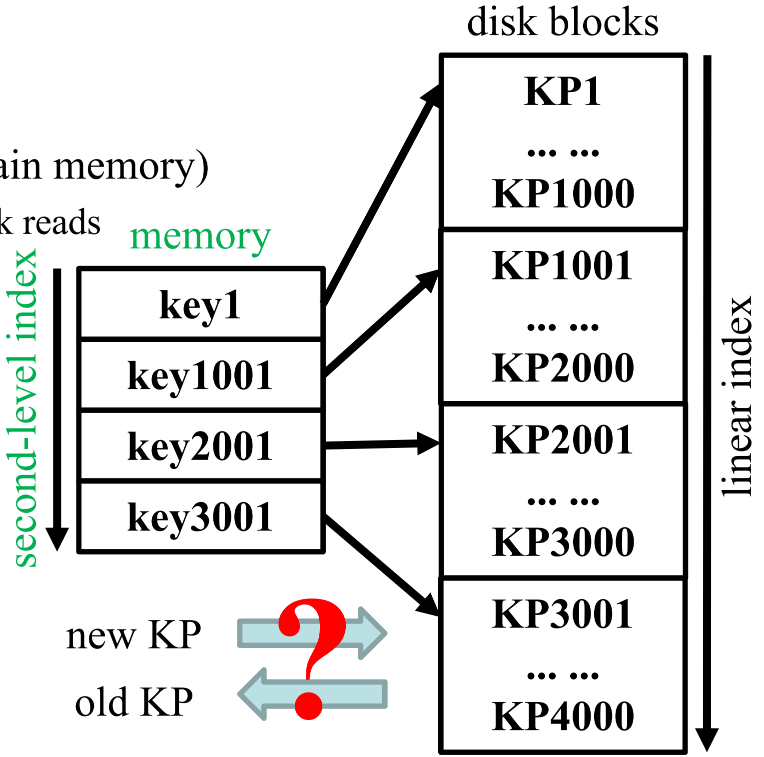
Indexing

- **Linear indexing**
  - fixed-length key & pointer pairs
    - automatic I/O of each entire block
  - linear index & second-level index (in main memory)
    - accessing a record requires (only) two disk reads
      - one from the linear index file
      - one from the original data file



Indexing

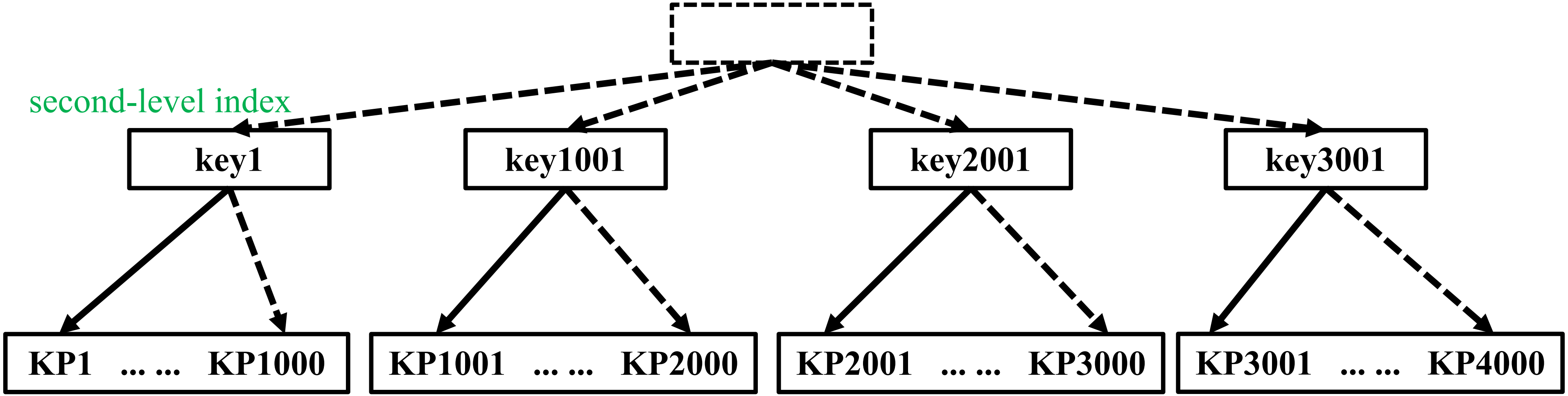
- **Linear indexing**
  - fixed-length key & pointer pairs
    - automatic I/O of each entire block
  - linear index & second-level index (in main memory)
    - accessing a record requires (only) two disk reads
      - one from the linear index file
      - one from the original data file
  - updates to a linear index are expensive
    - both insertion & removal are expensive
    - entire array contents might be shifted



Indexing

- **Linear indexing**
  - linear index & second-level index (in main memory)
  - *why updates to a linear index are expensive*?
    - perspective of *general tree*
    - the *tree* is so rigid, denying flexible & efficient updates

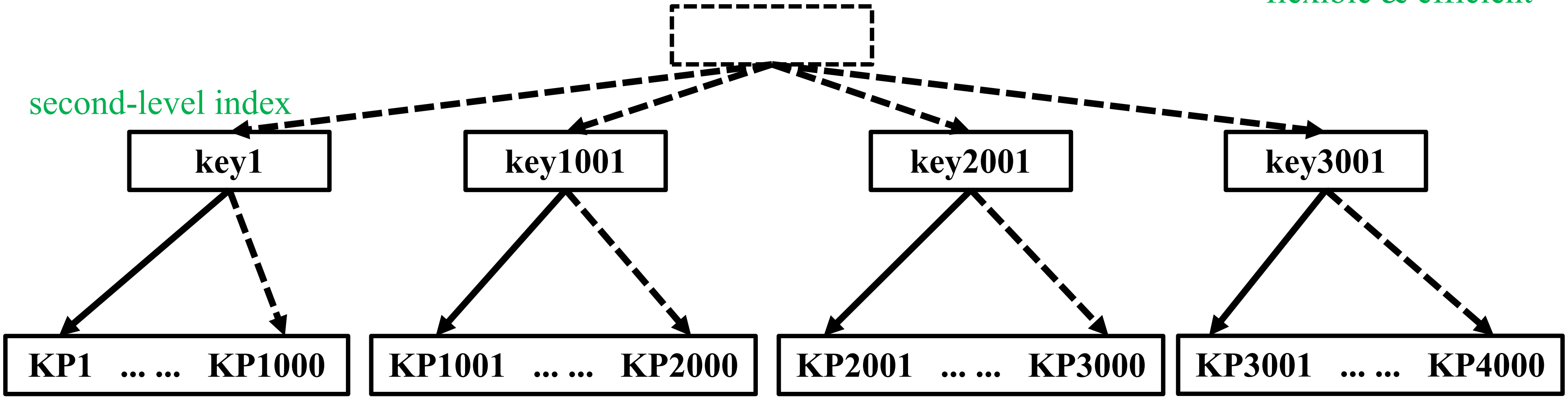
second-level index



Indexing

- **Linear indexing**
  - linear index & second-level index (in main memory)
  - *why updates to a linear index are expensive*?
    - perspective of *general tree*
    - the *tree* is so rigid, denying flexible & efficient updates

B-Tree
flexible & efficient



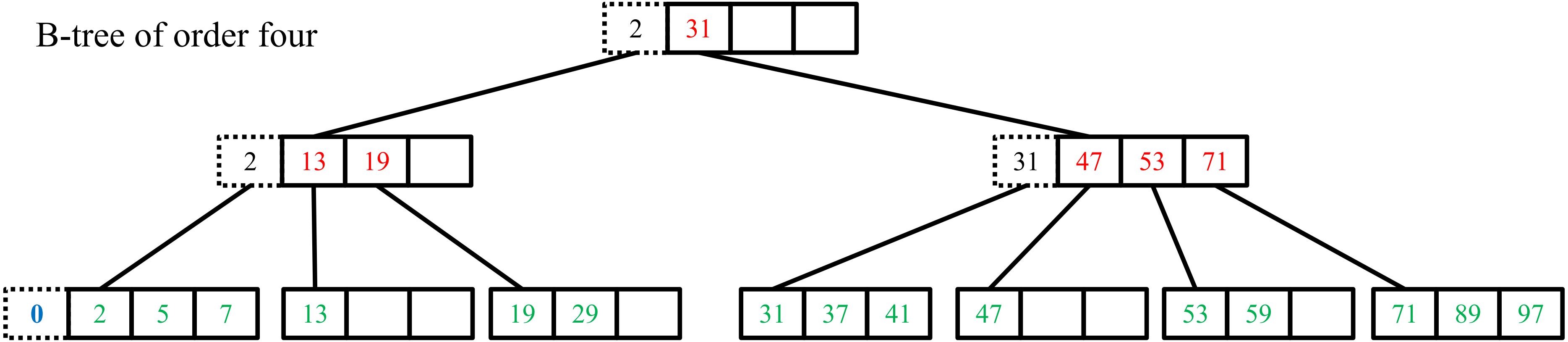
Indexing

- **B-Tree (of order *m*)**
  - root is either a leaf or has at least two children
  - internal node (except for root) has between $\lceil m/2 \rceil$ & *m* children
    - every node in the tree will be full at least to a certain minimum percentage
  - always *height balanced*, namely all leaves are at the same level
    - resonable distribution of records
  - internal node has *multiple keys* that separate its children
    - update & search operations affect only a few disk blocks & hence involve less disk I/O
    - keep related records i.e. records with similar key values on the same disk block, which helps to minimize I/O due to *locality of reference*
  - B-tree node is normally equivalent to a *disk block*

Indexing

- **B-Tree (of order *m*)**
  - internal node (except for root) has between $\lceil m/2 \rceil$ & *m* children
  - always *height balanced*, namely all leaves are at the same level
  - internal node has *multiple keys* that separate its children

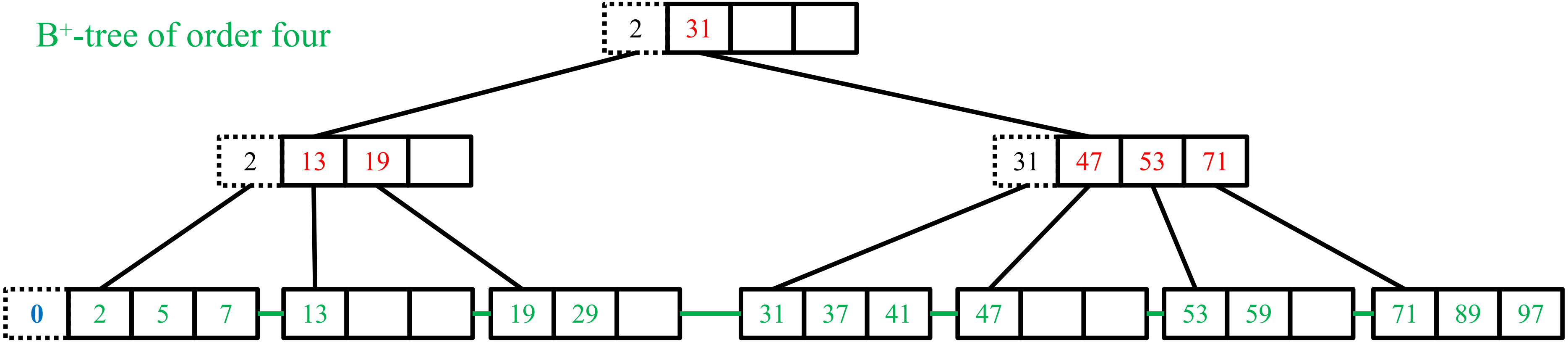
B-tree of order four



Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - internal node (except for root) has between $\lceil m/2 \rceil$ & m children
  - always *height balanced*, namely all leaves are at the same level
  - internal node has *multiple keys* that separate its children
  - store records (or record pointers, primary key pointers) only at leaves

B+-tree of order four

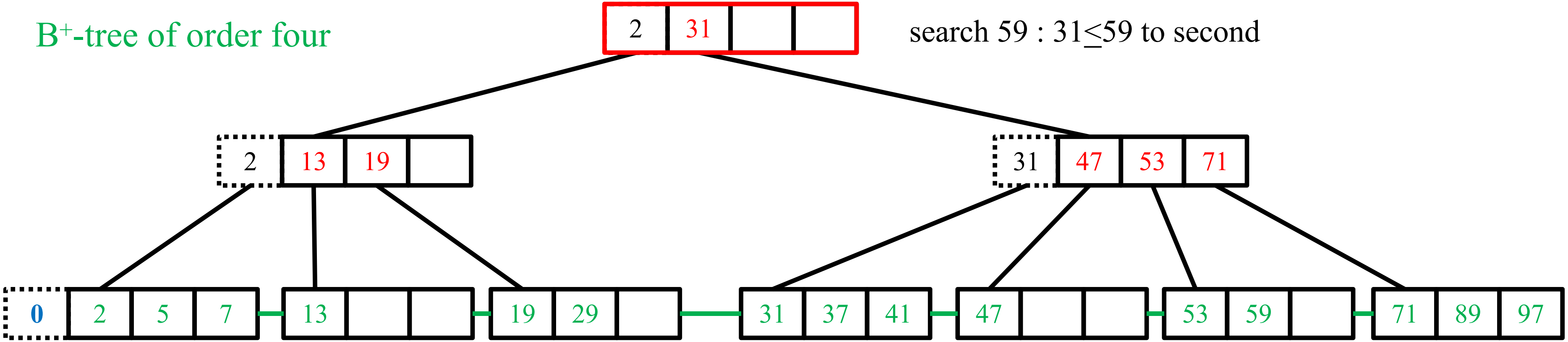


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
    - *binary search* may be applied within each node (block)
    - e.g. search 59

B+-tree of order four

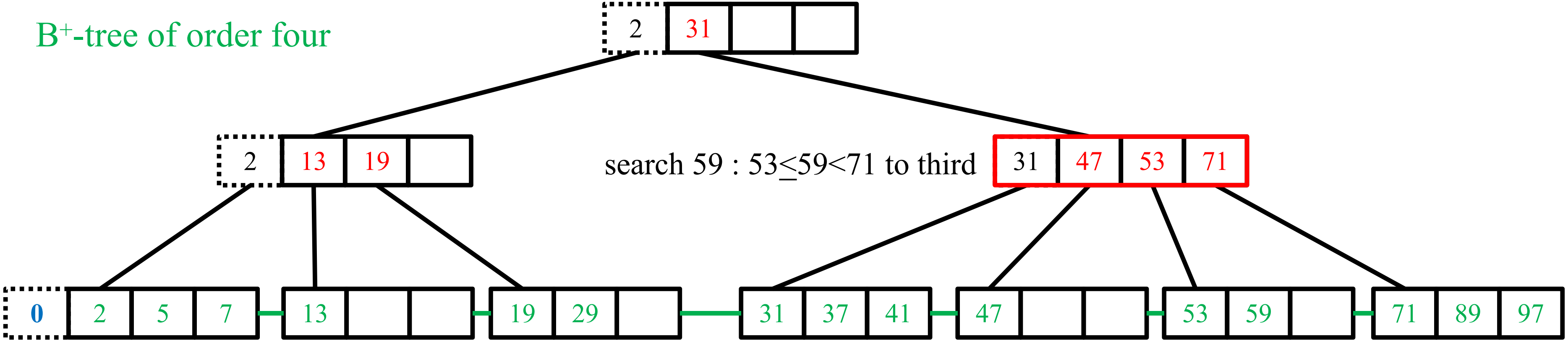


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
    - *binary search* may be applied within each node (block)
    - e.g. search 59

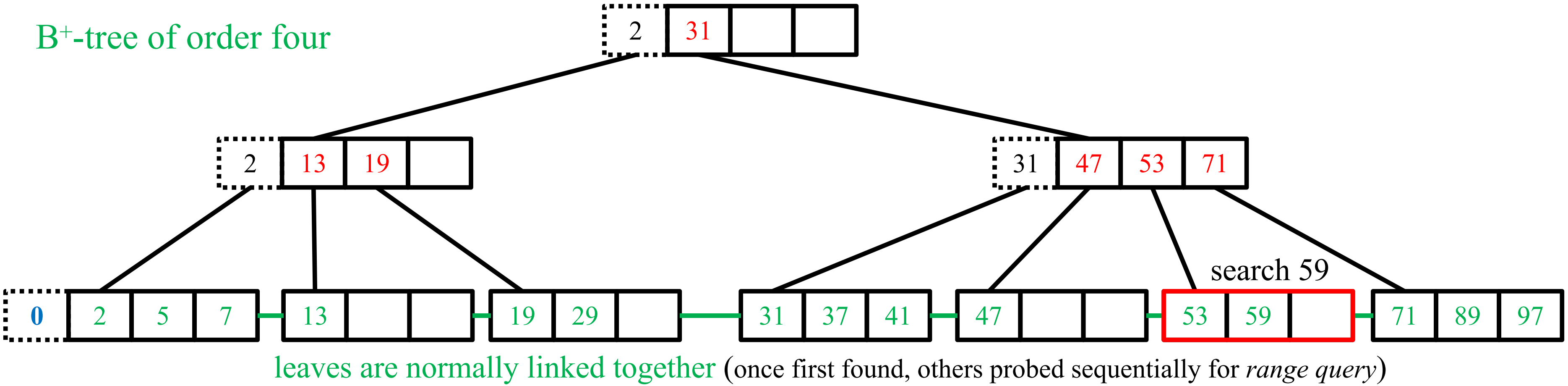
B$^+$-tree of order four



leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

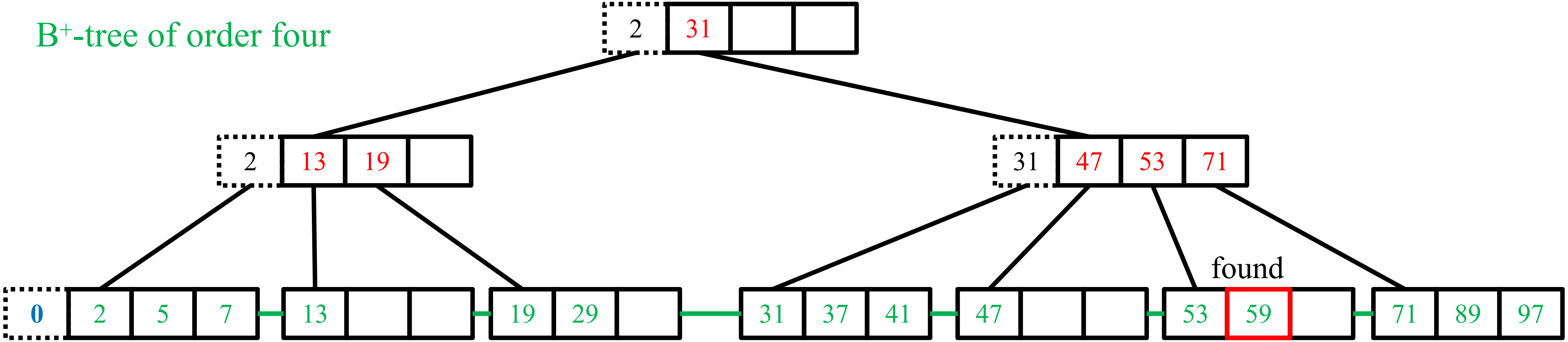
- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
    - *binary search* may be applied within each node (block)
    - e.g. search 59



Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
    - *binary search* may be applied within each node (block)
    - e.g. search 59

B$^+$-tree of order four



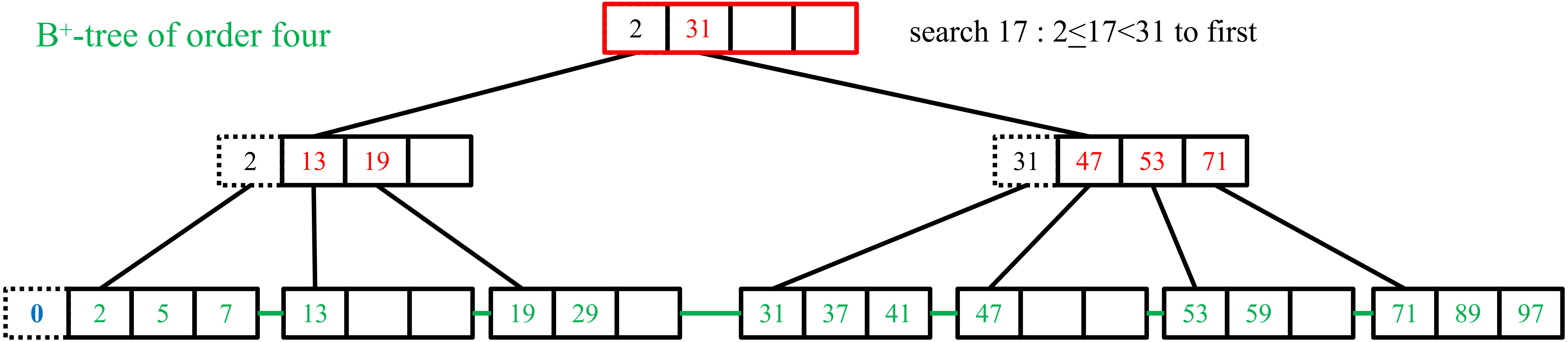
leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
    - *binary search* may be applied within each node (block)
    - e.g. search 59, search 17

B⁺-tree of order four

search 17 : 2≤17<31 to first

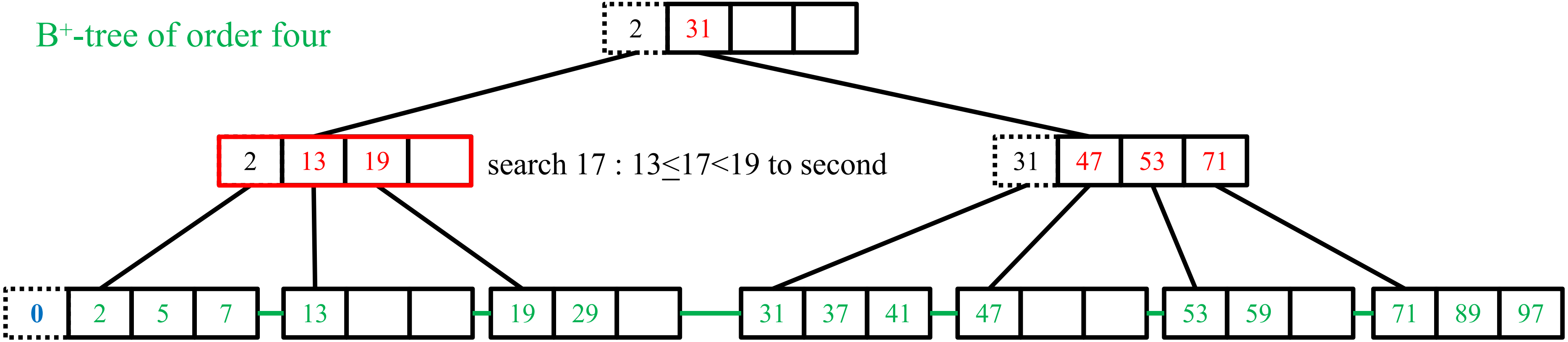


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
    - *binary search* may be applied within each node (block)
    - e.g. search 59, search 17

B+-tree of order four

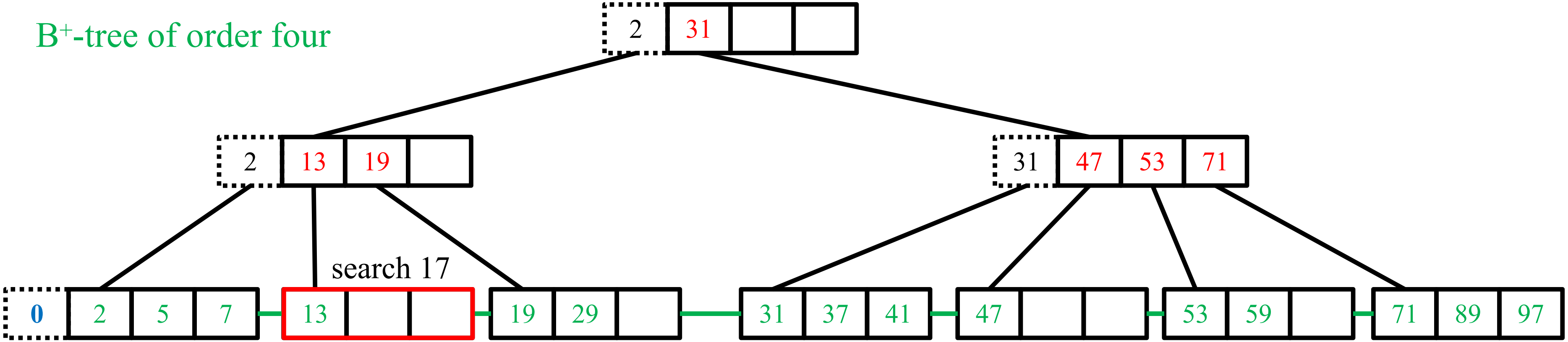


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
    - *binary search* may be applied within each node (block)
    - e.g. search 59, search 17

B⁺-tree of order four

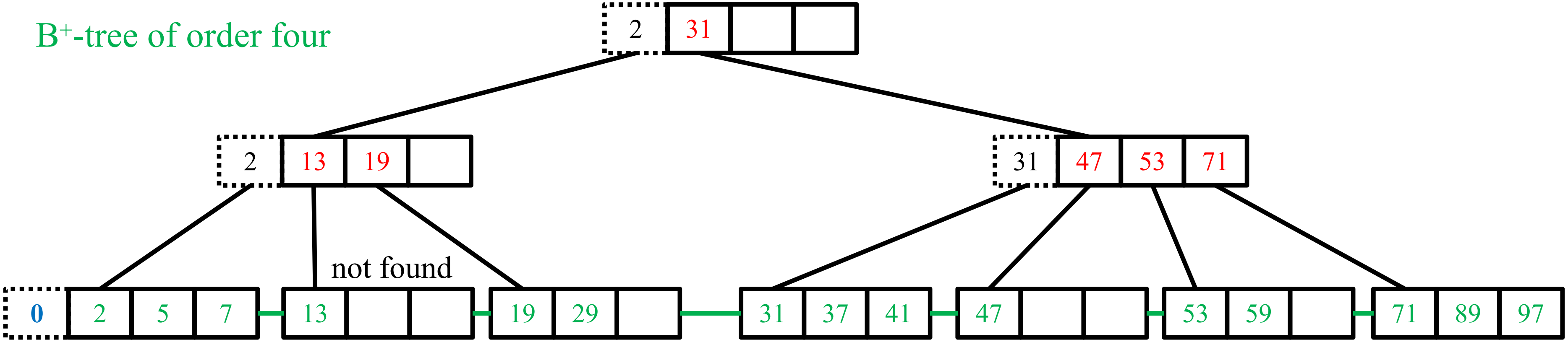


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
    - *binary search* may be applied within each node (block)
    - e.g. search 59, search 17

B⁺-tree of order four

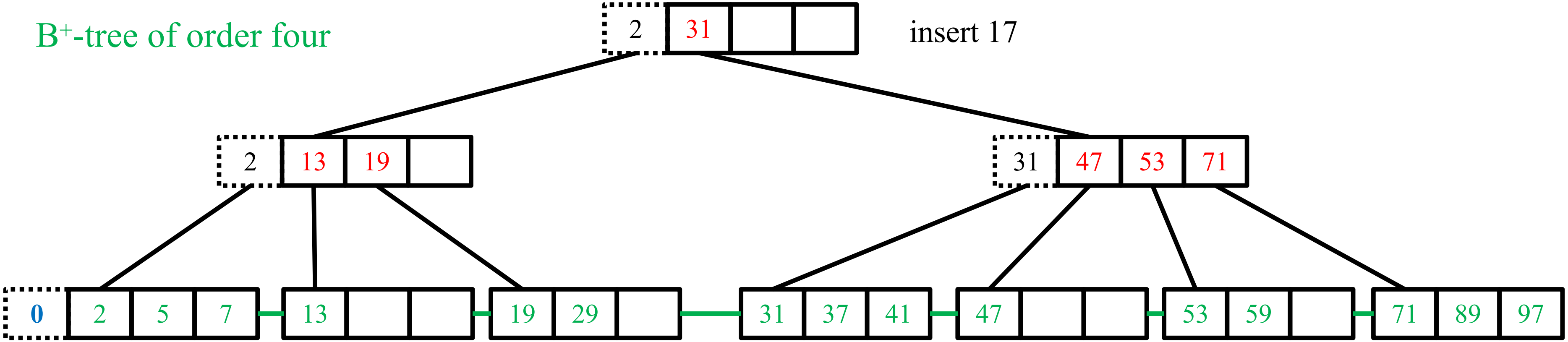


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B⁺-tree node is affected

B⁺-tree of order four

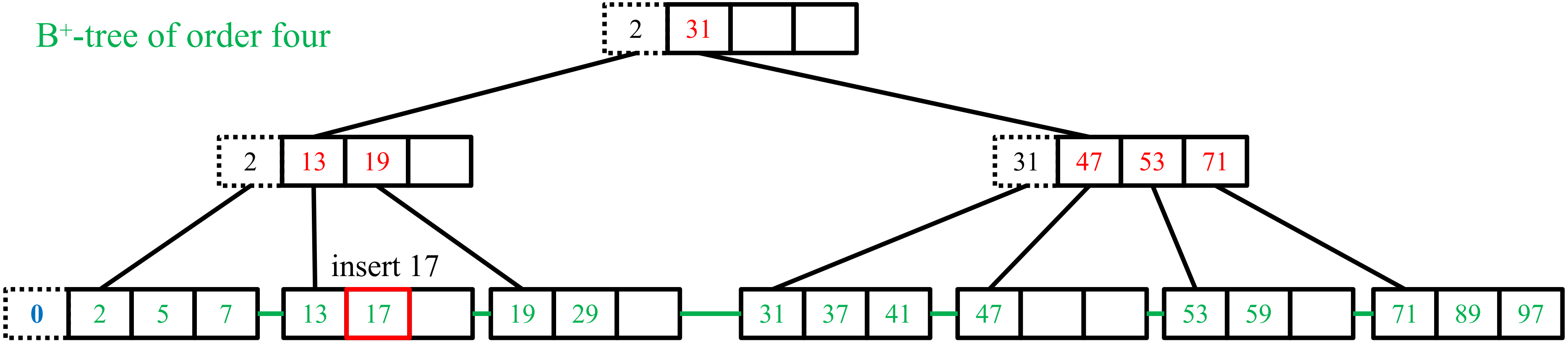


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B⁺-tree node is affected

B⁺-tree of order four



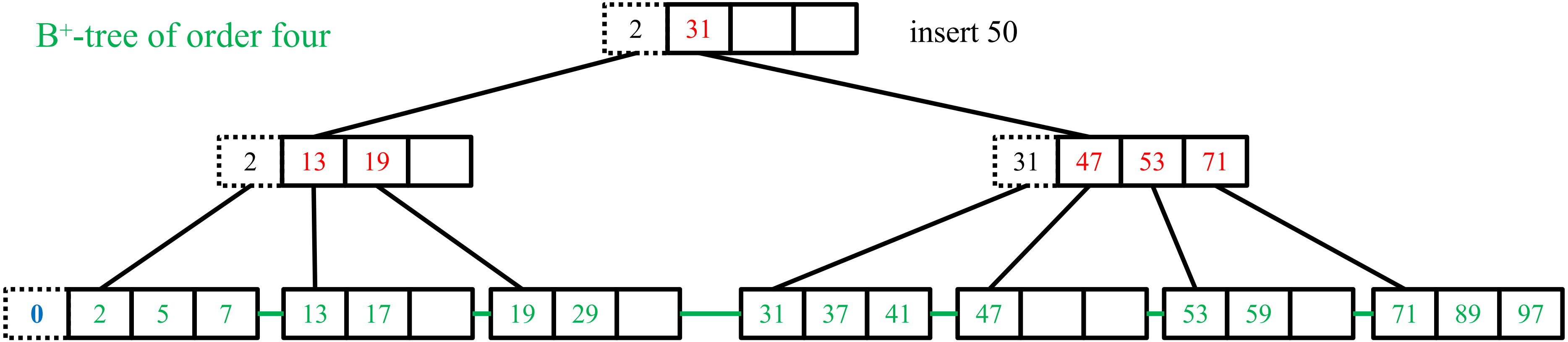
leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B⁺-tree node is affected

B⁺-tree of order four

insert 50

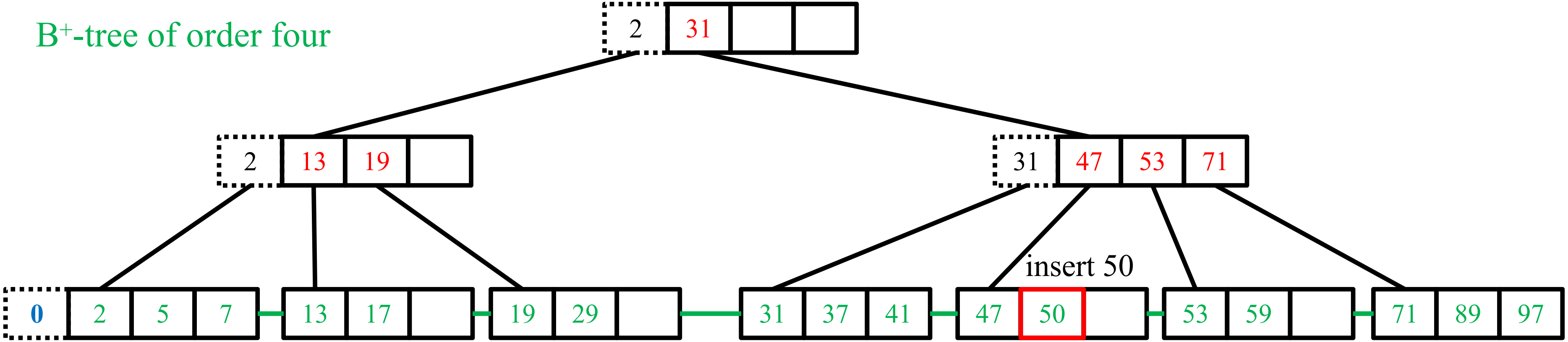


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B⁺-tree node is affected

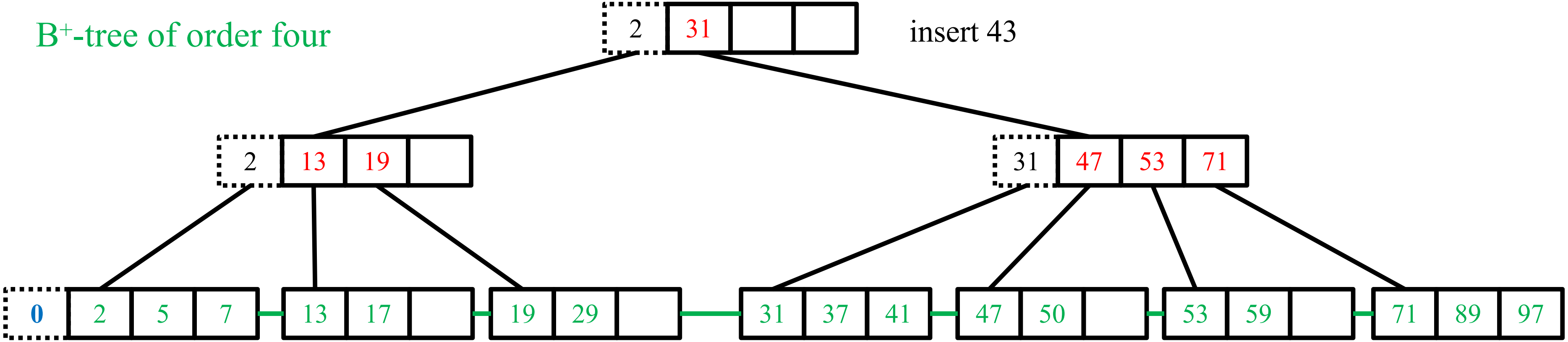
B⁺-tree of order four



leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B+-tree node is affected
    - if leave is full, perform **splitting-promotion**
      - split the node by *middle* & promote *middle* to parent

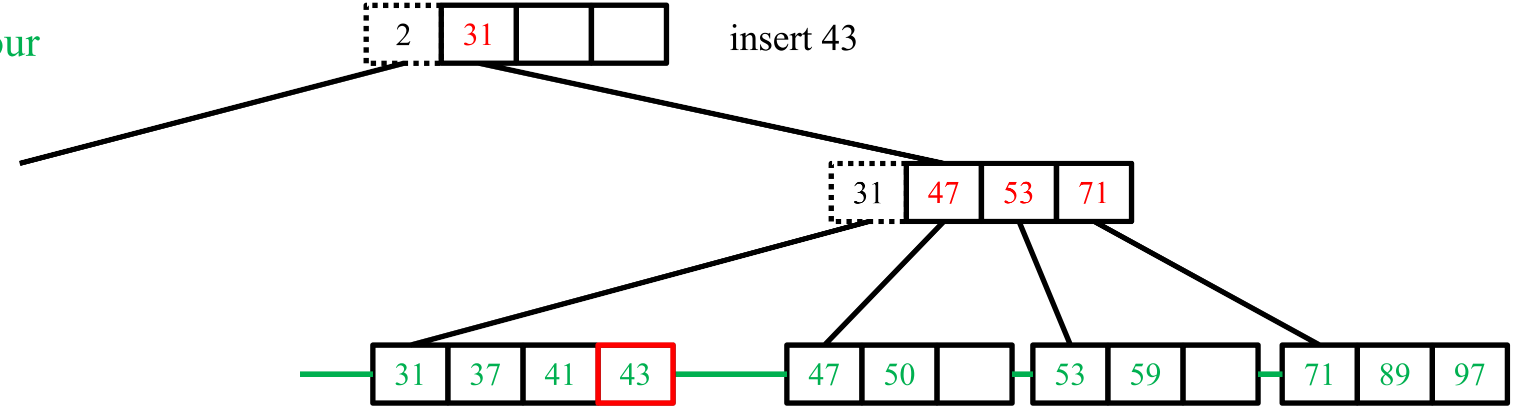


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B$^+$-tree node is affected
    - if leave is full, perform **splitting-promotion**
      - split the node by *middle* & promote *middle* to parent

B$^+$-tree of order four

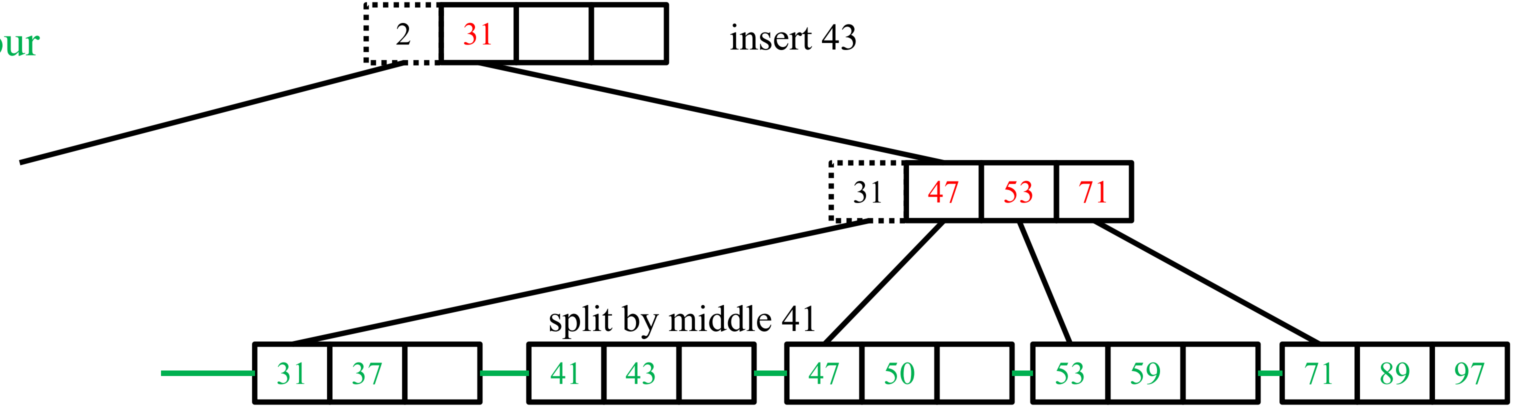


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B⁺-tree node is affected
    - if leave is full, perform **splitting-promotion**
      - split the node by *middle* & promote *middle* to parent

B⁺-tree of order four

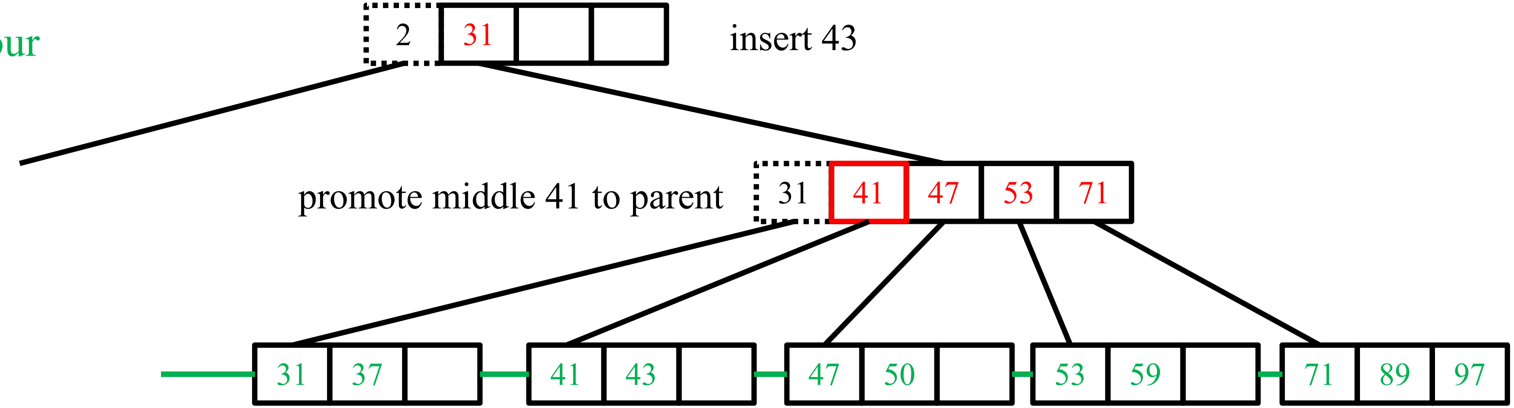


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B+-tree node is affected
    - if leave is full, perform **splitting-promotion**
      - split the node by *middle* & promote *middle* to parent

B+-tree of order four

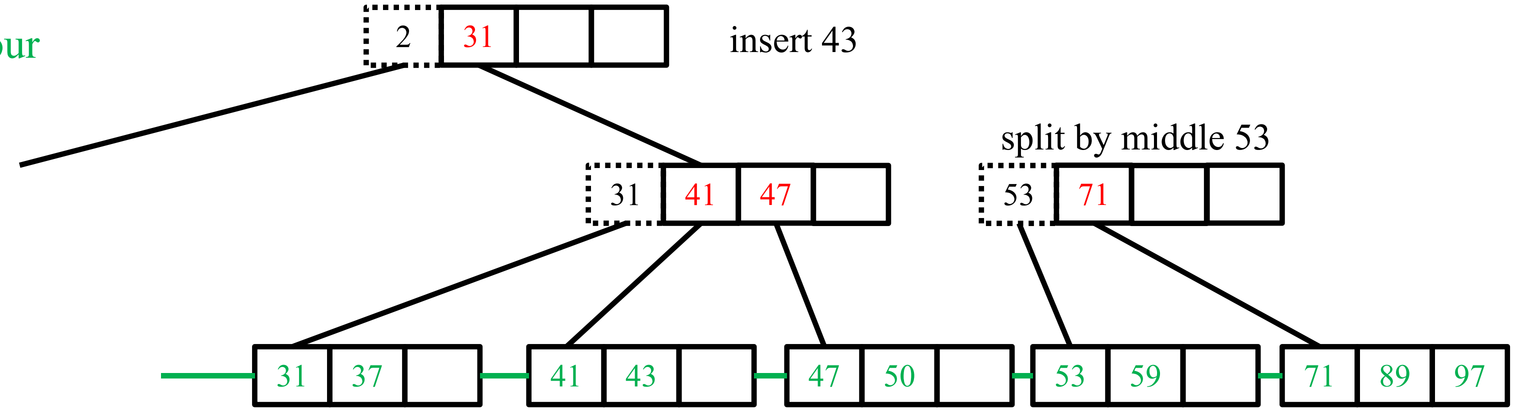


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B⁺-tree node is affected
    - if leave is full, perform **splitting-promotion**
      - split the node by *middle* & promote *middle* to parent

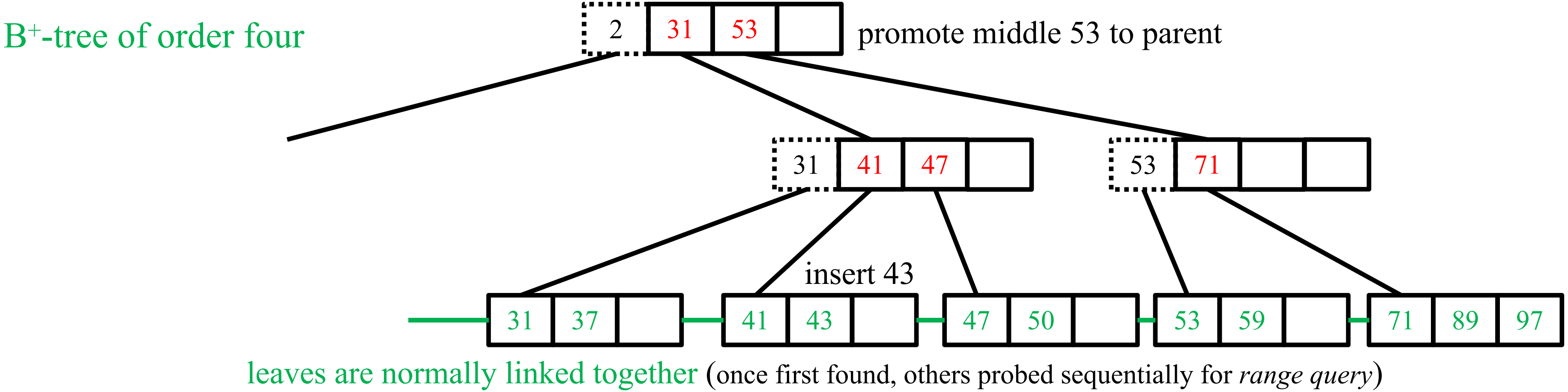
B⁺-tree of order four



leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

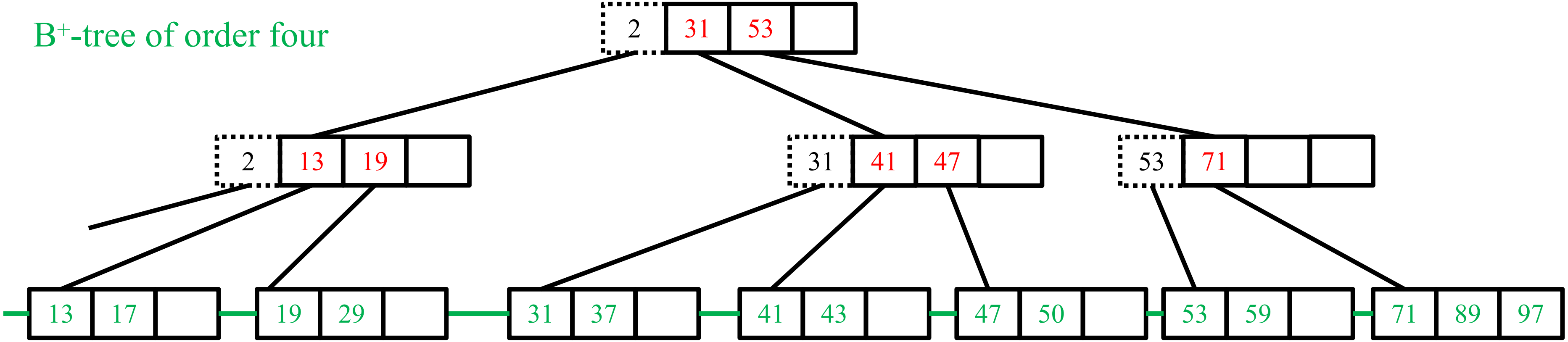
- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B+-tree node is affected
    - if leave is full, perform **splitting-promotion**
      - split the node by *middle* & promote *middle* to parent



Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert
    - if leave is not full, just insert into the leave; no other B$^+$-tree node is affected
    - if leave is full, perform **splitting-promotion**
    - such insertion process is guaranteed to **keep all nodes at least half full**

B$^+$-tree of order four

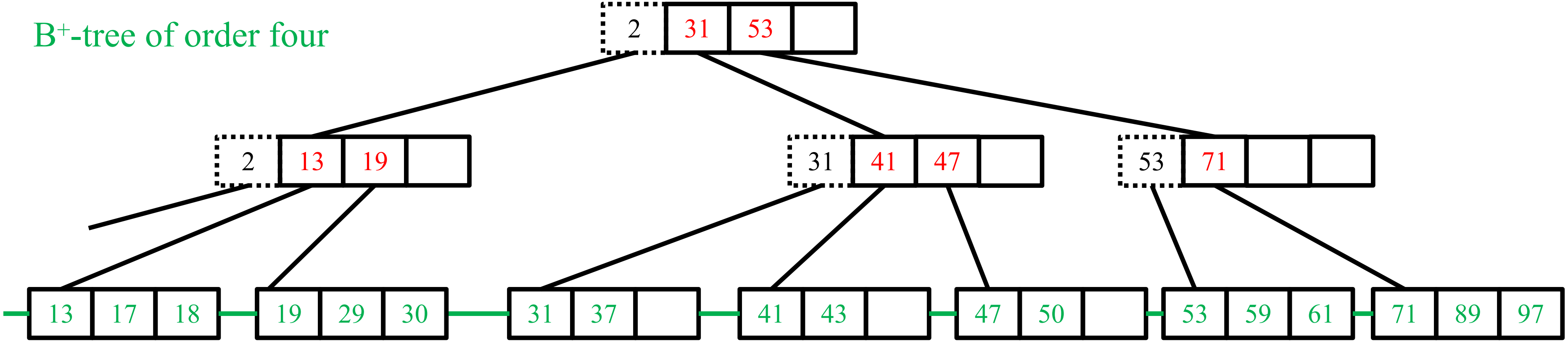


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - suppose all nodes are intended to be kept at least half full (similar logic applies for other minimum thresholds)

B$^+$-tree of order four

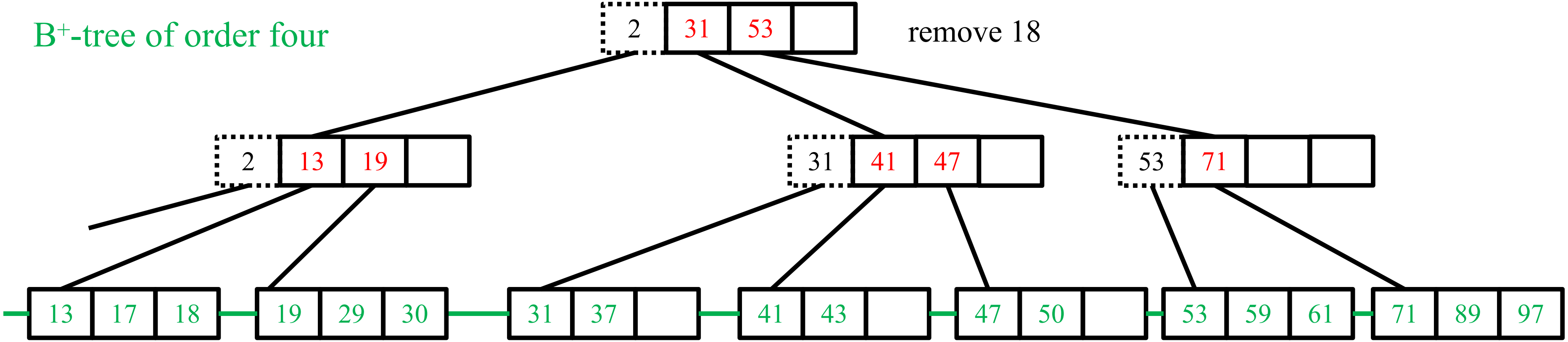


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave

B$^+$-tree of order four

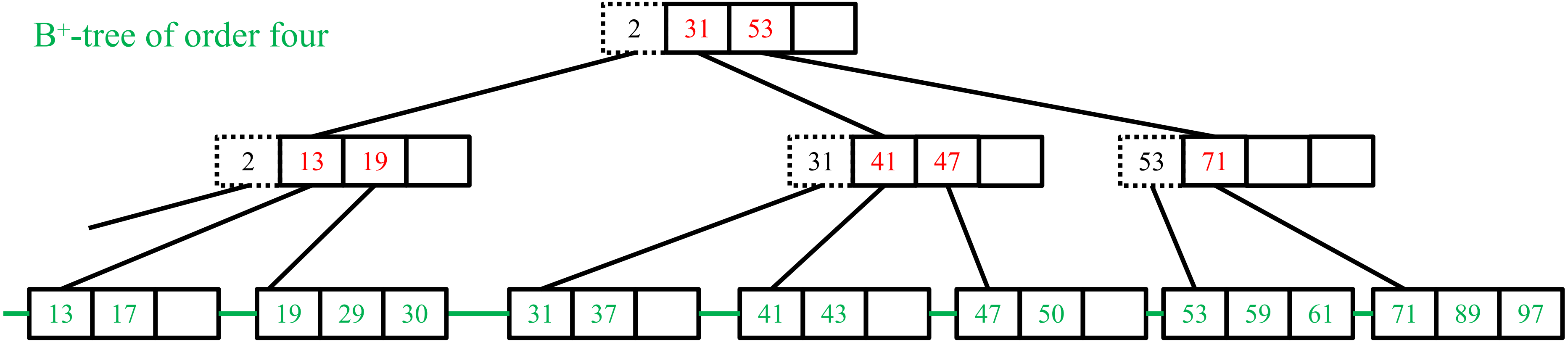


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave

B+-tree of order four

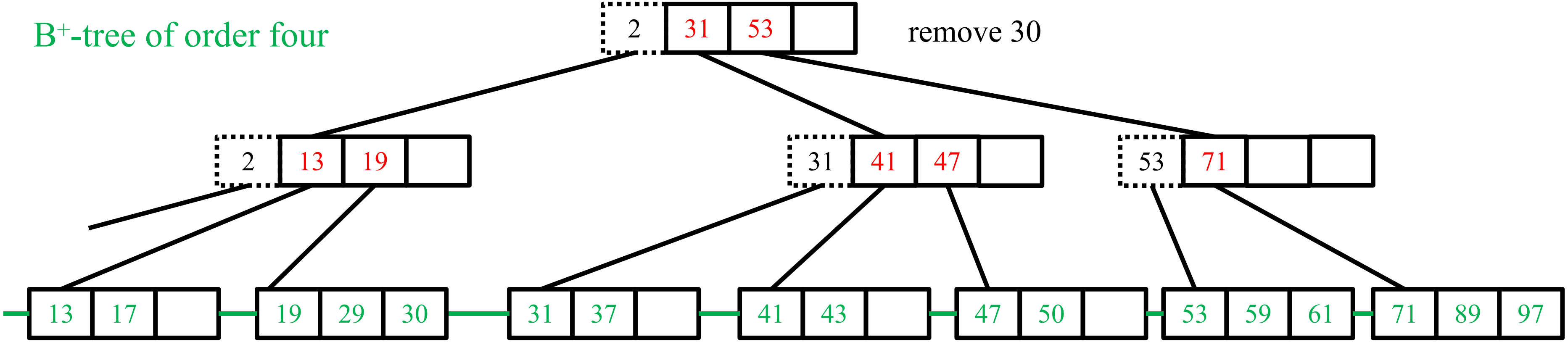


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave

B⁺-tree of order four

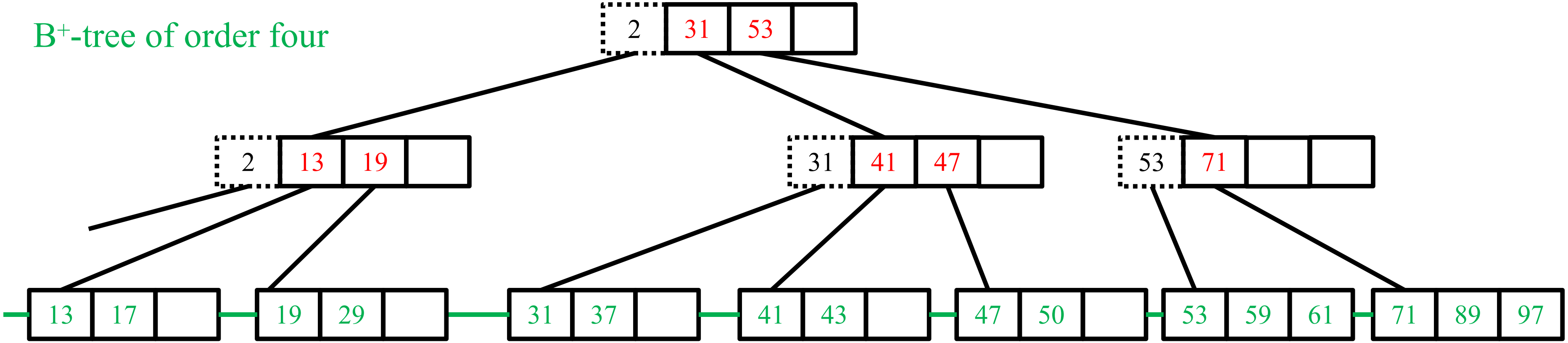


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B^+-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave

B^+-tree of order four



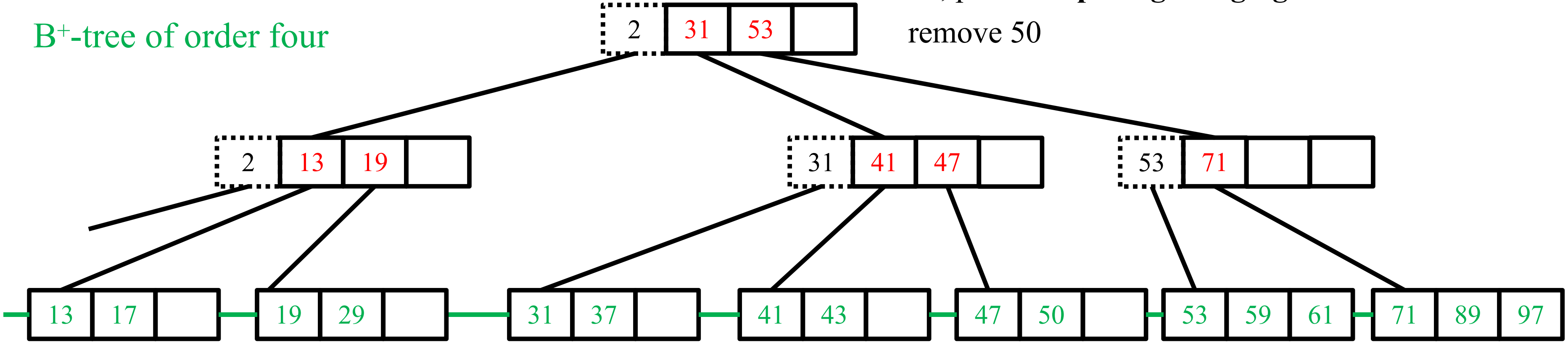
leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B⁺-tree of order four

remove 50

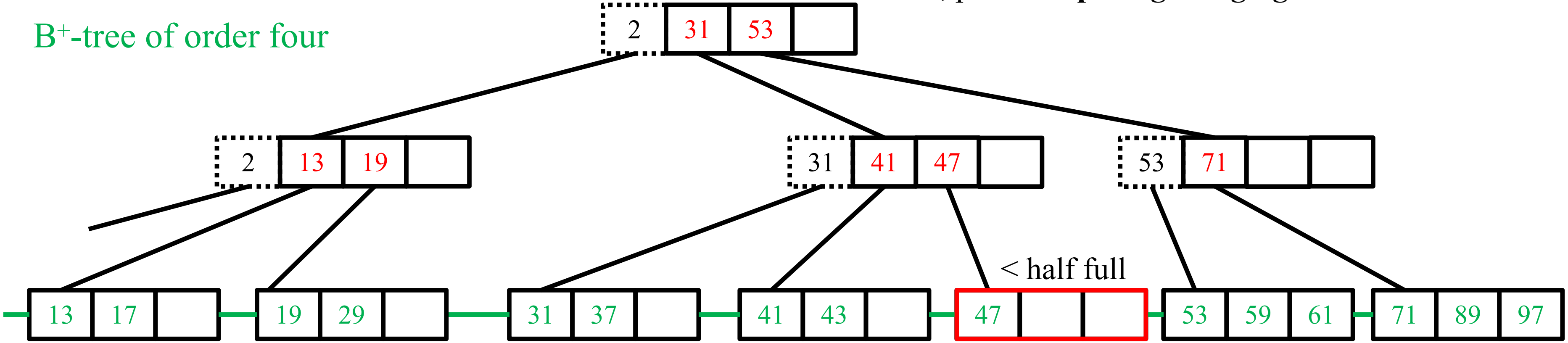


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B$^+$-tree of order four

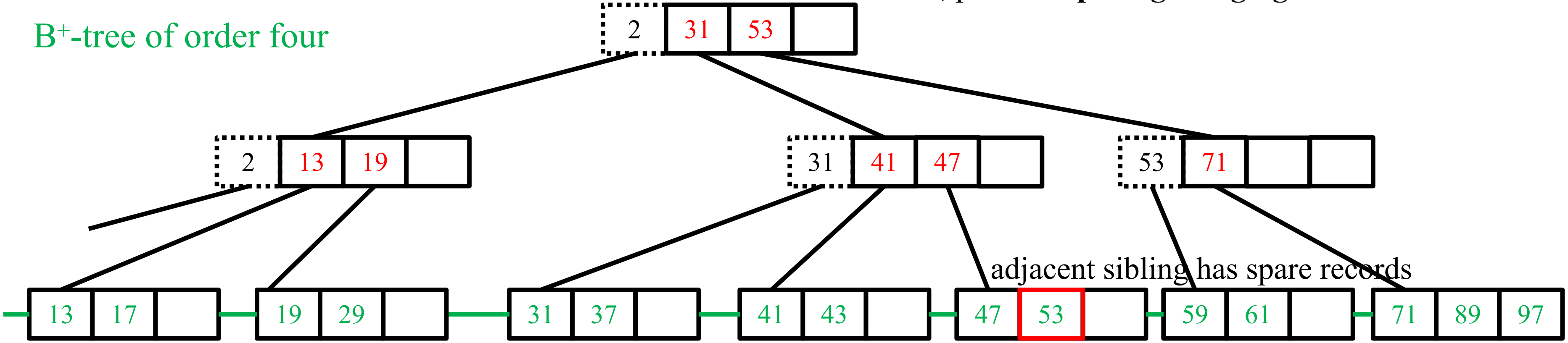


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B+-tree of order four



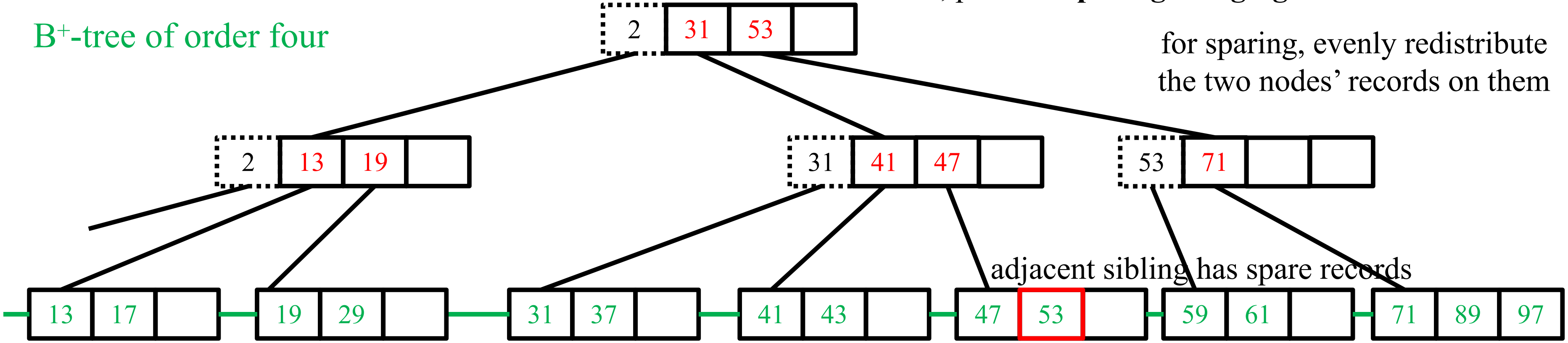
leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B+-tree of order four

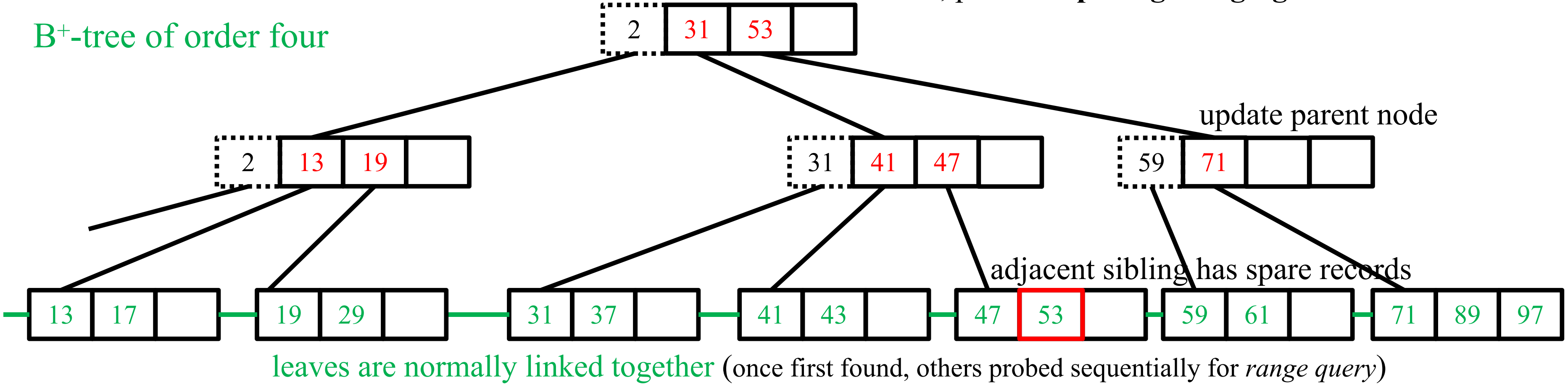
for sparing, evenly redistribute the two nodes' records on them



leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

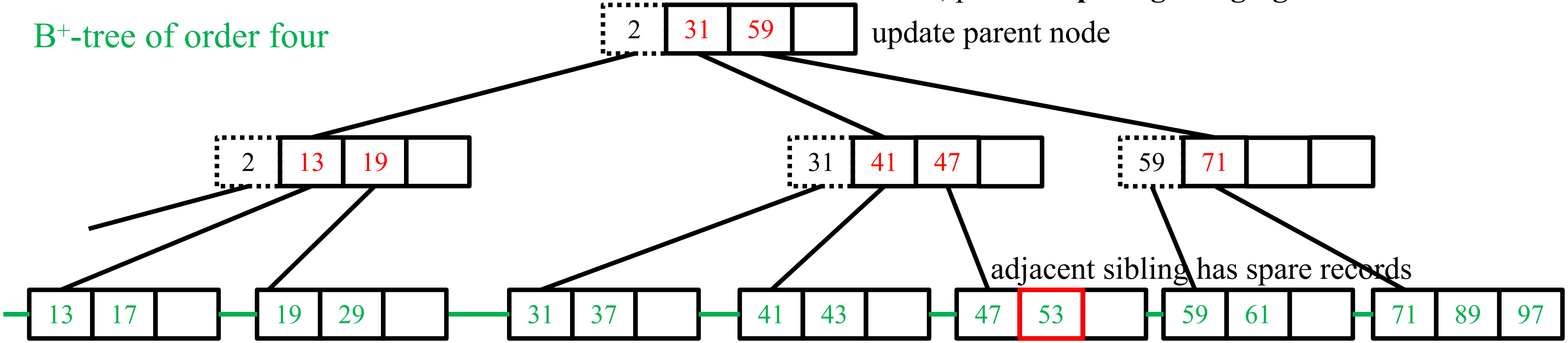
- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**



Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B$^+$-tree of order four



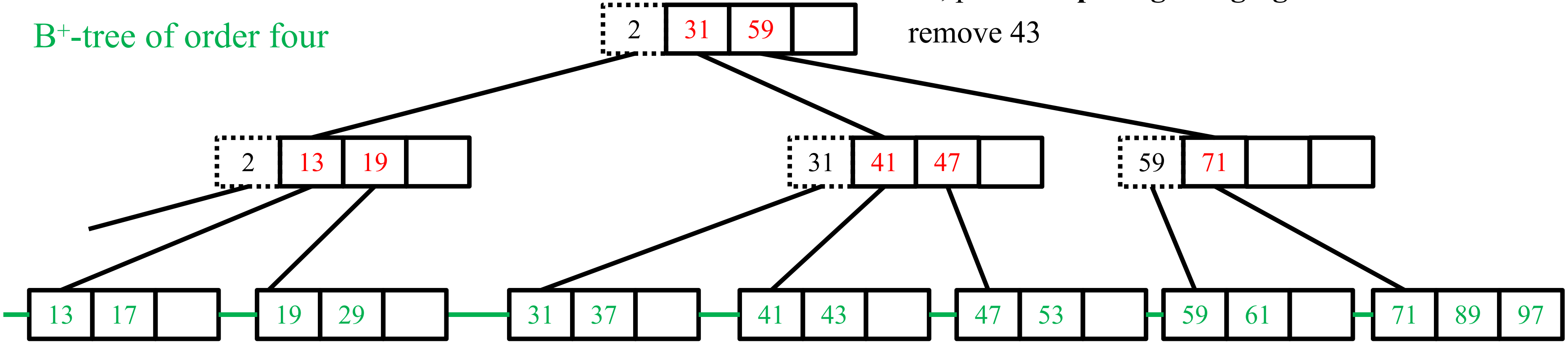
leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B$^+$-tree of order four

remove 43

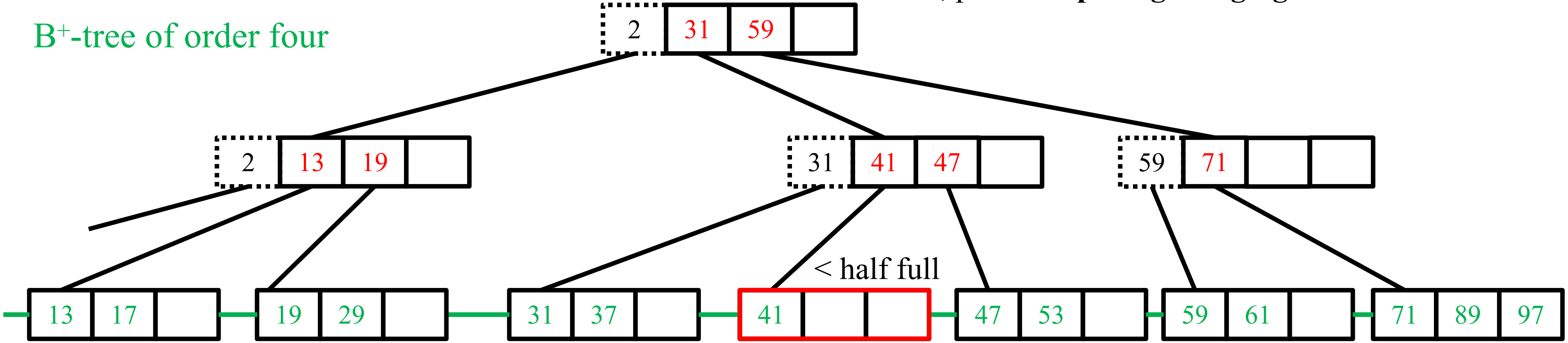


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B⁺-tree of order four

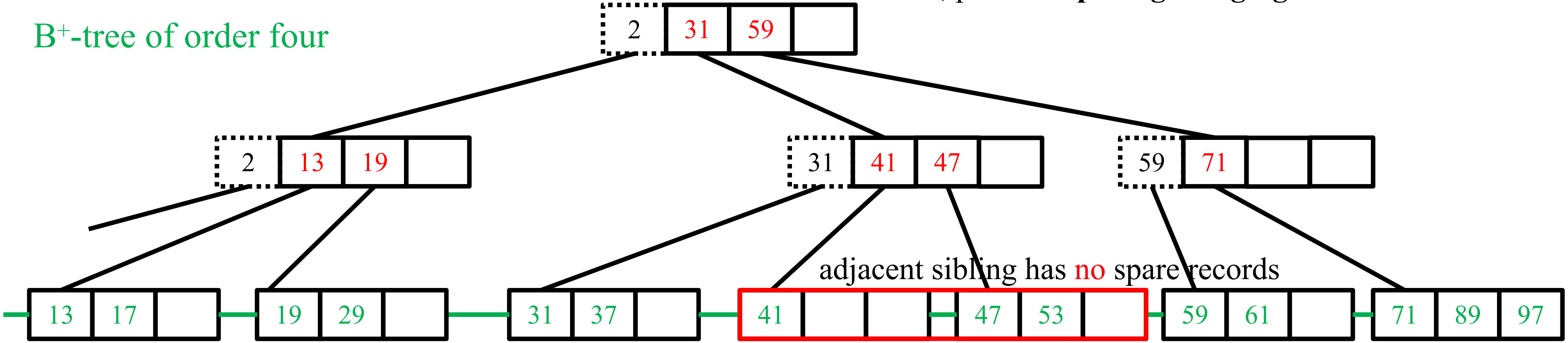


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B+-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B+-tree of order four

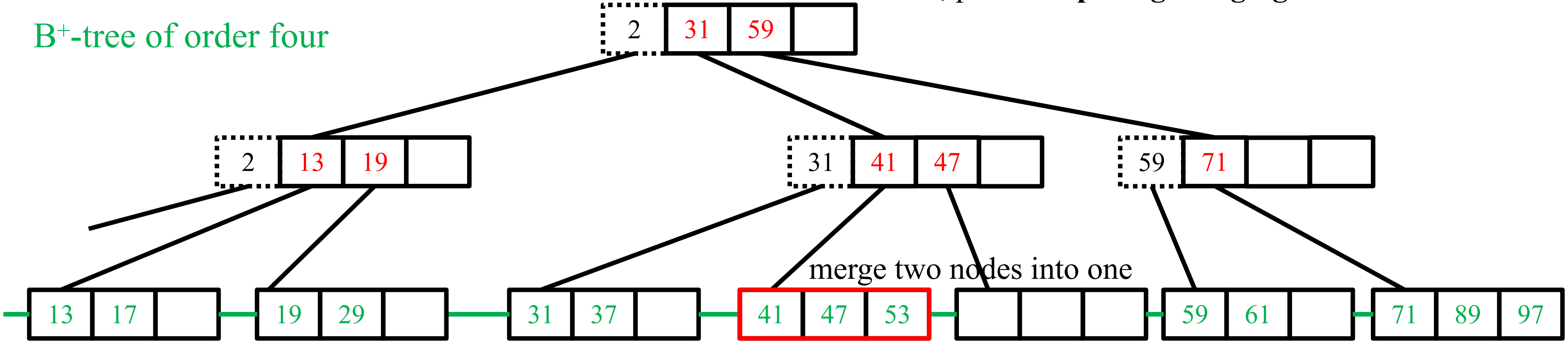


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B⁺-tree of order four

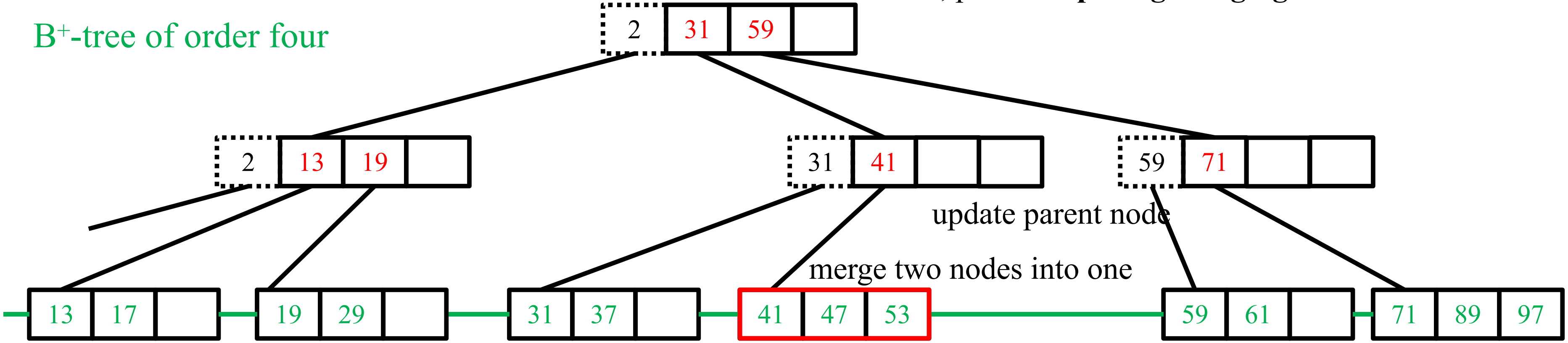


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

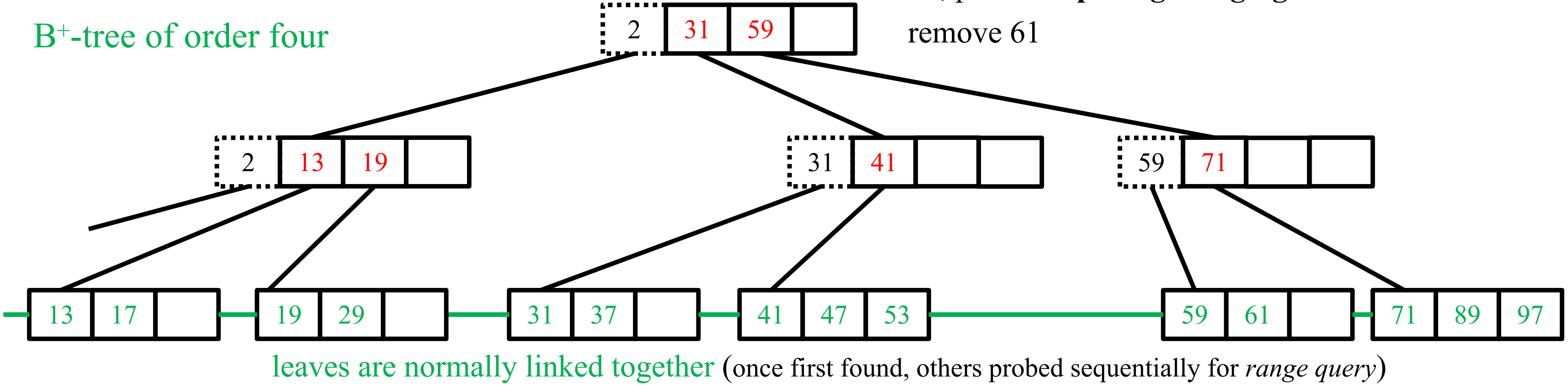
B⁺-tree of order four



leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

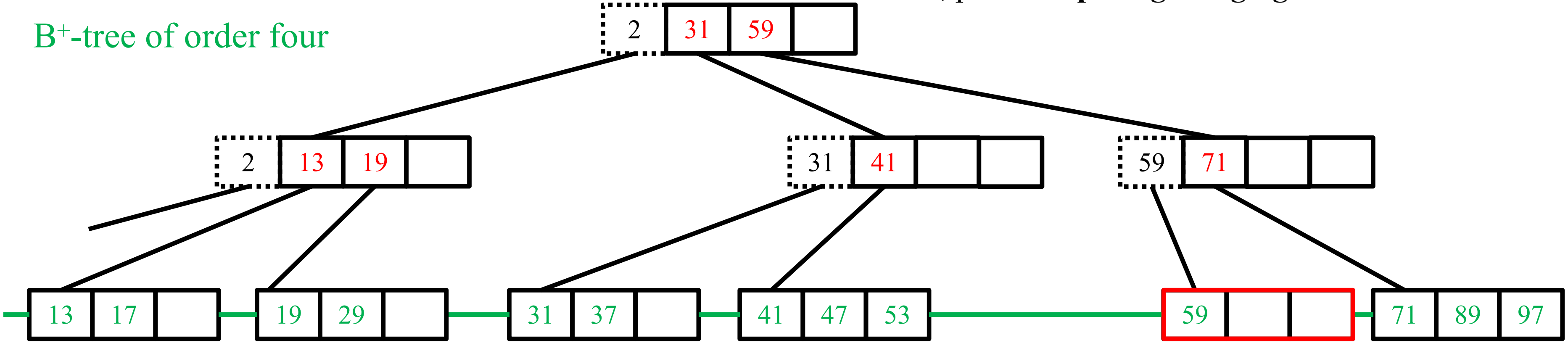
- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**



Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B⁺-tree of order four

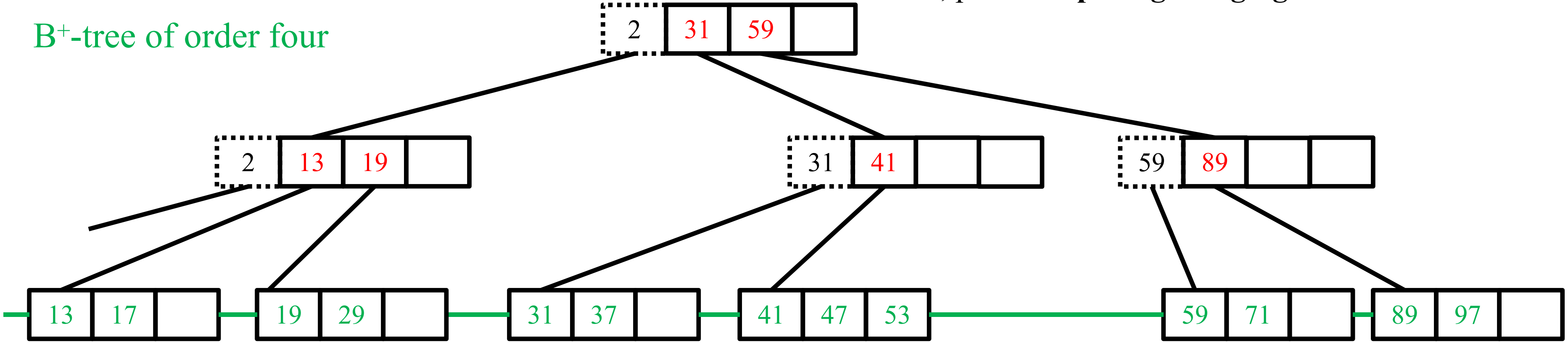


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B$^+$-tree of order four



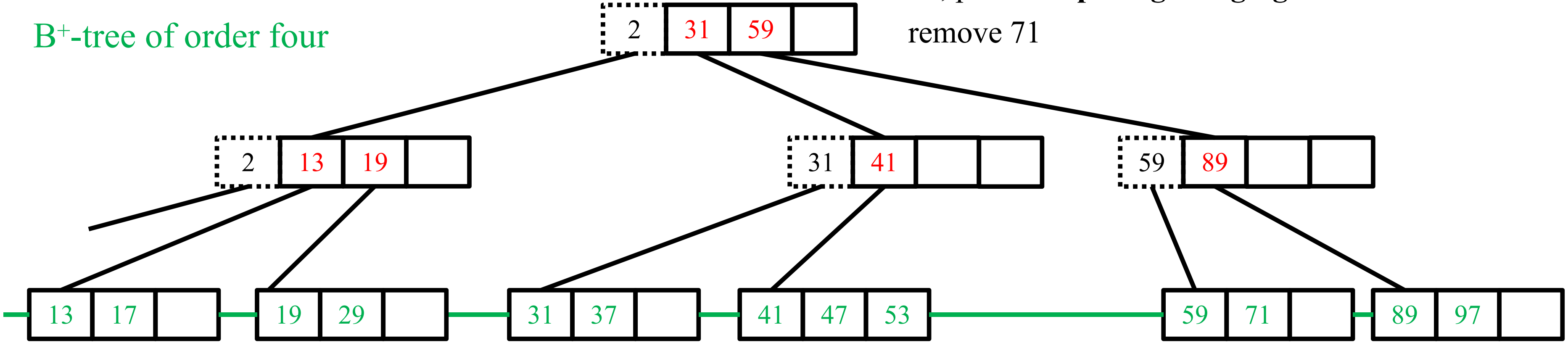
leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B⁺-tree of order four

remove 71

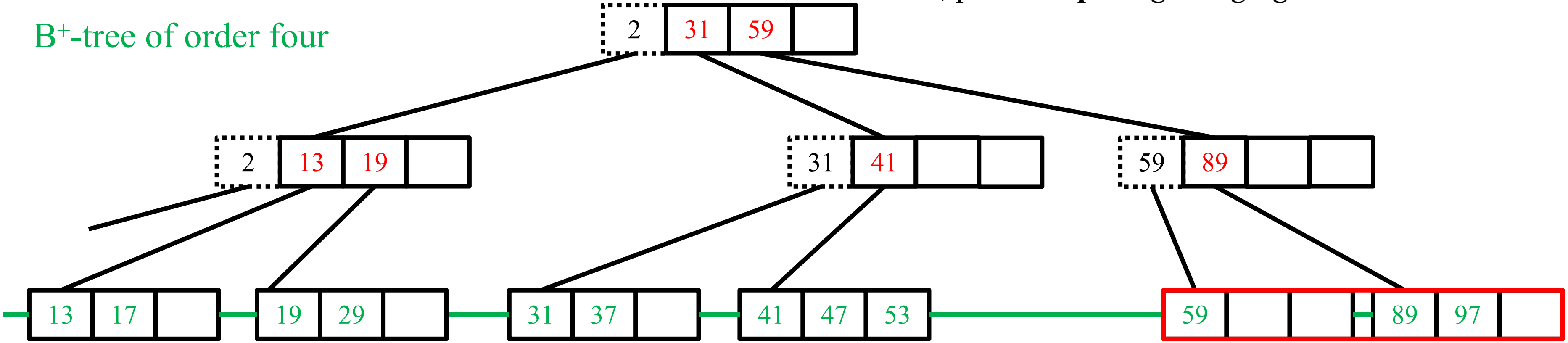


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B$^+$-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B$^+$-tree of order four

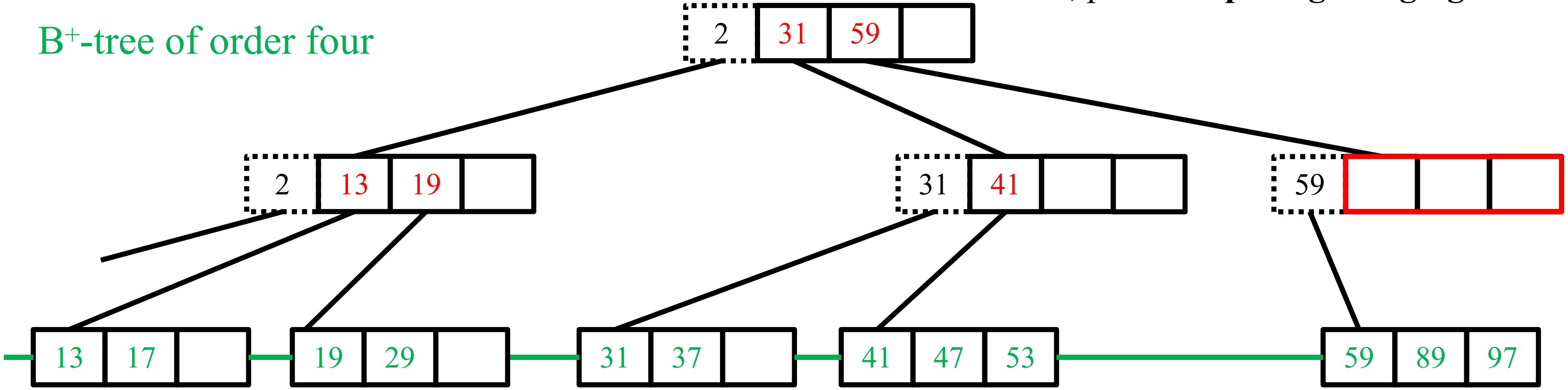


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B⁺-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B⁺-tree of order four

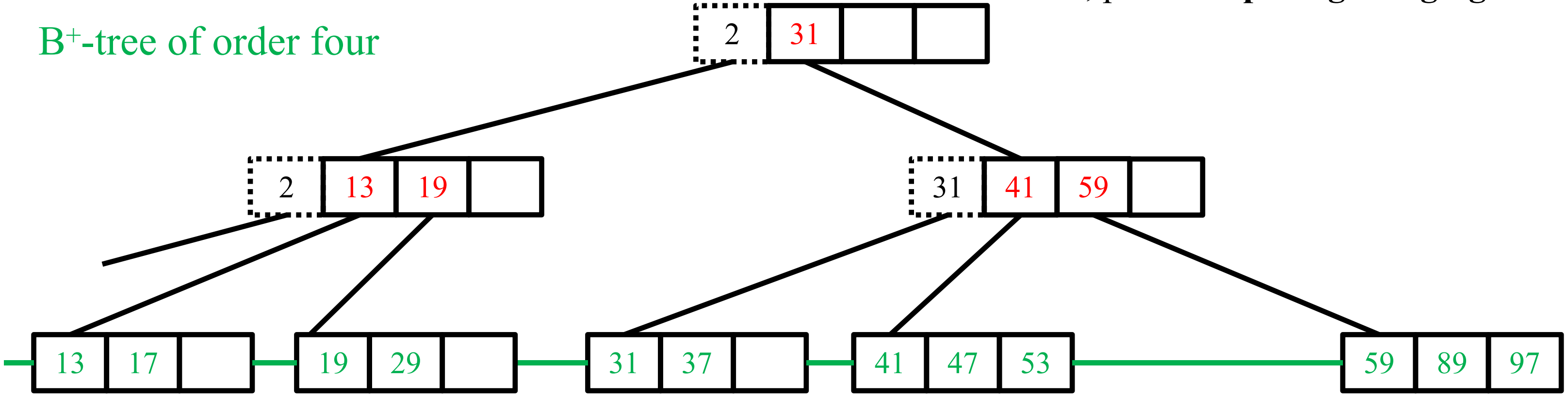


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B^{+}-Tree (of order *m*) - actually adopted in practice**
  - search
  - insert - splitting-promotion
  - remove
    - if leave can be at least half full after removal, just remove from the leave
    - if leave is less than half full after removal, perform **sparing-merging**

B^{+}-tree of order four

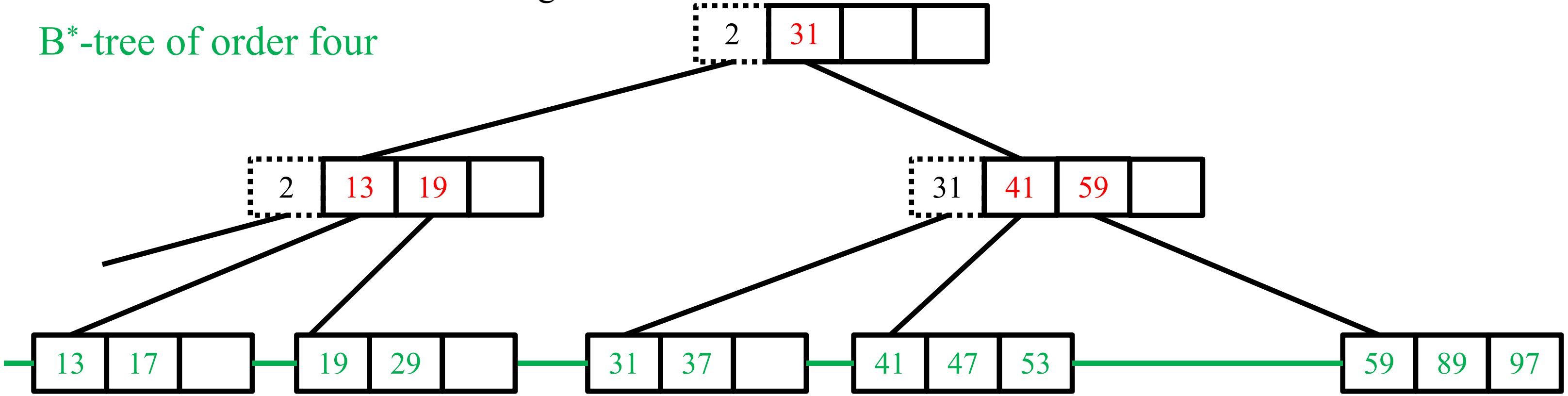


leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B*-Tree (of order *m*) - actually adopted in practice**
  - insert - splitting-promotion
  - remove - sparing-merging
  - identical to B+-tree, except for rules of splitting & merging nodes
    - split two nodes into three instead of one into two
    - merge three nodes into two instead of two into one

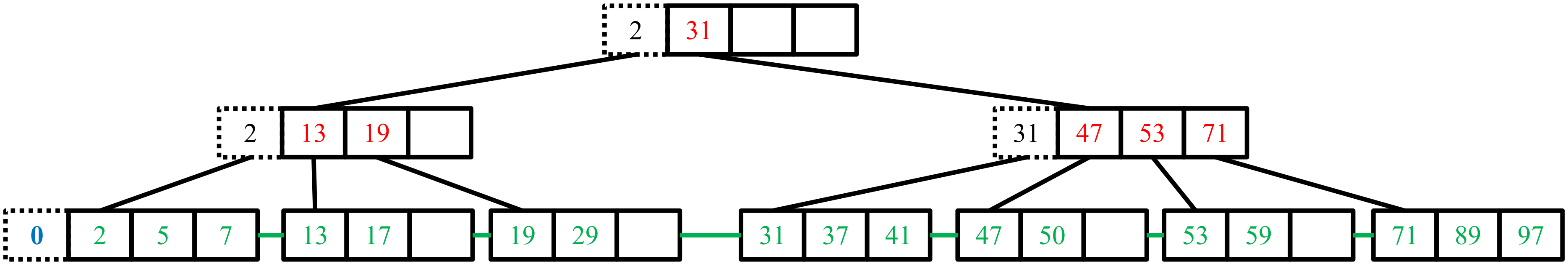
B*-tree of order four



leaves are normally linked together (once first found, others probed sequentially for *range query*)

Indexing

- **B-Tree family (of order *m*) - B$^+$/B*-tree actually adopted in practice**
  - internal node (except for root) has between $\lceil m/2 \rceil$ & *m* children
  - always *height balanced*, namely all leaves are at the same level
  - internal node has *multiple keys* that separate its children
  - store records (or record pointers, primary key pointers) only at leaves (B$^+$/B*-tree)
    - insertion process is guaranteed to **keep all nodes at least half full**
  - order *m* is large (e.g. 100) in practice, so overhead for internal nodes is very low



leaves are normally linked together (once first found, others probed sequentially for *range query*)

THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY